Howard Chen (SBN 257393)
howard.chen@klgates.com
Harold H. Davis, Jr. (SBN 235552)
harold.davis@klgates.com
Rachel Davidson (SBN 215517)
rachel.davidson@klgates.com
K&L GATES LLP
Four Embarcadero Center, Suite 1200
San Francisco, California 94111
Telephone: 415.882.8200
Facsimile: 415.882.8220

Attorneys for Plaintiffs
WISTRON CORPORATION,
AOPEN INCORPORATED AND
AOPEN AMERICA INCORPORATED

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

SAN FRANCISCO DIVISION

WISTRON CORPORATION, a Taiwan corporation, AOPEN INCORPORATED, a Taiwan corporation, AOPEN AMERICA INCORPORATED, a California corporation, Plaintiffs, vs. PHILLIP M. ADAMS & ASSOCIATES, LLC, a Utah limited liability corporation, Defendant.	Case No. ____________ **COMPLAINT FOR DECLARATORY JUDGMENT OF NON-INFRINGEMENT AND INVALIDITY OF U.S. PATENT NOS. 5,379,414; 5,983,002; 6,195,767; AND 6,401,222**

Plaintiffs WISTRON CORPORATION ("Wistron"), AOPEN INCORPORATED ("AOpen Inc."), and AOPEN AMERICA INCORPORATED ("AOpen America") (collectively "Wistron and AOpen") by and through their attorneys allege as follows:

1. This is a civil action arising under the patent laws of the United States, 35 U.S.C. §§ 101, et seq., seeking a declaratory judgment, under 28 U.S.C. §§ 2201 and 2202, that no valid and enforceable claim of United States Patent No. 5,379,414, (the "'414 patent"), United States Patent No. 5,983,002 (the "'002 patent"), United States Patent No. 6,195,767 (the "'767 patent"), and United States Patent No. 6,401,222 (the "'222 patent") (collectively the "patents-in-suit") is infringed by Wistron and AOpen. This Complaint further seeks a declaratory judgment that the patents-in-suit are invalid as anticipated under 35 U.S.C. § 102, invalid as obvious under 35 U.S.C. § 103, and invalid for failure to meet the requirements of 35 U.S.C § 112.

THE PARTIES

2. Plaintiff Wistron is a Taiwan corporation with its principal place of business at 21F, 88, Sec. 1, Hsin Tai Wu. Rd., Hsichih, Taipei Hsien 221, Taiwan, R.O.C. Wistron is one of the world's largest original design manufacturers ("ODM") for information and communication technology ("ICT") products. Wistron is in the business of designing, developing and manufacturing computer products such as notebook computers, computer peripheral equipment and other electronic products for customers to sell under their own brand name. Wistron products are sold throughout the United States, including this District.

3. Plaintiff AOpen Inc. is a Taiwan corporation with its principal place of business at No.68, Ruiguang Rd., Neihu District, Taipei, Taiwan, ROC. AOpen Inc. is in the business of manufacturing and selling computer products such as notebook computers and computer peripheral equipment. AOpen products are sold throughout the United States, including this District.

4. Plaintiff AOpen America is a California corporation with its principal place of business at 2890 Zanker Road, Suite 101 San Jose, CA 95134.

5. Defendant Phillip M. Adams & Associates, L.L.C. ("Adams") is a Utah limited liability company with an address at 325 Federal Heights Circle, Salt Lake City, Utah 84103. Adams has an alternative address at P.O. Box 1207, Bountiful, Utah 84011.

JURISDICTION AND VENUE

6. Wistron and AOpen file this Complaint against Adams pursuant to the patent laws of the United States, Title 35 of the United States Code, with a specific remedy sought based upon the laws authorizing actions for declaratory judgment in the federal courts of the United States, 28 U.S.C. §§ 2201 and 2202.

7. This Court has subject matter jurisdiction over this action, which arises under the patent laws of the United States, pursuant to 28 U.S.C. §§ 1331, 1338(a), and under the Federal Declaratory Judgment Act, 28 U.S.C. §§ 2201 and 2202.

8. Personal jurisdiction and venue are proper in this District pursuant to 28 U.S.C. §§ 1391 and 1400(b). Upon information and belief, Adams conducts business in this District, Wistron and AOpen do business in this District, and a substantial part of the events that give rise to the action occurred in this District. Upon information and belief, Adams has and continues to transact business in this District by providing consulting services, negotiating licensing arrangements, and participating in litigation in and directed at companies located in this District.

INTRADISTRICT ASSIGNMENT

9. This action is properly filed in the San Francisco Division of the Northern District of California because Wistron, AOpen and Adams do business within the San Francisco Division.

EXISTENCE OF AN ACTUAL CONTROVERSY

10. There is an actual controversy within the jurisdiction of this Court under 28 U.S.C. §§ 2201 and 2202.

11. On or around June 21, 2010, in a letter sent to Wistron and AOpen, respectively, Adams embarked upon an improper campaign of threats against Wistron and AOpen to file a baseless patent infringement lawsuit against Wistron and AOpen for the purpose, and intended effect, of disrupting the sale of Wistron and AOpen notebook computers in the United States.

12. Adams has repeatedly demanded that Wistron and AOpen enter into a royalty-bearing license for the patents-in-suit. Adams is claiming that certain Wistron and AOpen products infringe one or more claims of the patents-in-suit, and has told Wistron and AOpen that if they do not take a license to the patents-in-suit, Wistron and AOpen may be subject to substantial liabilities.

13. As part of its improper campaign, Adams referred Wistron and AOpen to the previously-filed *Lenovo* case, Case No. 1:05-cv-64 TX-DN (D. Utah), in which infringement of the patents-in-suit were alleged against other computer manufacturers and computer brand companies such as Lenovo, Dell, Asus, Fujitsu and the like.

14. On or around August 20, 2010, Adams sent another letter to Wistron and AOpen, respectively, alleging that Wistron and AOpen manufactured infringing products and that Wistron and AOpen were independently liable for patent infringement for the patents-in-suit.

15. Based upon the above facts, there is an actual and justiciable controversy within the jurisdiction of this Court under 28 U.S.C. §§ 2201 and 2202.

FIRST CLAIM

DECLARATORY JUDGMENT REGARDING THE '414 PATENT

16. Wistron and AOpen hereby restate and reallege the allegations set forth in paragraphs 1 through 15 and incorporate them by reference.

17. Phillip M. Adams is the inventor of the '414 patent, entitled "Systems and Methods for FDC Error Detection and Prevention." A true and correct copy of the '414 patent is attached hereto as Exhibit A.

18. Wistron and AOpen seek a judicial determination and declaration that no valid and enforceable claim of the '414 patent is infringed by Wistron and AOpen.

19. Wistron and AOpen seek a judicial determination and declaration that the '414 patent is invalid because it fails to satisfy the conditions and requirements for patentability as set forth, *inter alia*, in Sections 101, 102, 103, and/or 112 of Title 35 of the United States Code.

SECOND CLAIM

DECLARATORY JUDGMENT REGARDING THE '002 PATENT

20. Wistron and AOpen hereby restate and reallege the allegations set forth in paragraphs 1 through 19 and incorporate them by reference.

21. Phillip M. Adams is the inventor of the '002 patent, entitled "Defective Floppy Diskette Controller Detection Apparatus and Method." A true and correct copy of the '002 patent is attached hereto as Exhibit B.

22. Wistron and AOpen seek a judicial determination and declaration that no valid and enforceable claim of the '002 patent is infringed by Wistron and AOpen.

23. Wistron and AOpen seek a judicial determination and declaration that the '002 patent is invalid because it fails to satisfy the conditions and requirements for patentability as set forth, *inter alia*, in Sections 101, 102, 103, and/or 112 of Title 35 of the United States Code.

THIRD CLAIM

DECLARATORY JUDGMENT REGARDING THE '767 PATENT

24. Wistron and AOpen hereby restate and reallege the allegations set forth in paragraphs 1 through 23 and incorporate them by reference.

25. Phillip M. Adams is the inventor of the '767 patent, entitled "Data Corruption Detection Apparatus and Method." A true and correct copy of the '767 patent is attached hereto as Exhibit C.

26. Wistron and AOpen seek a judicial determination and declaration that no valid and enforceable claim of the '767 patent is infringed by Wistron and AOpen.

27. Wistron and AOpen seek a judicial determination and declaration that the '767 patent is invalid because it fails to satisfy the conditions and requirements for patentability as set forth, *inter alia*, in Sections 101, 102, 103, and/or 112 of Title 35 of the United States Code.

FOURTH CLAIM

DECLARATORY JUDGMENT REGARDING THE '222 PATENT

28. Wistron and AOpen hereby restate and reallege the allegations set forth in paragraphs 1 through 27 and incorporate them by reference.

29. Phillip M. Adams is the inventor of the '222 patent, entitled "Defective Floppy Diskette Controller Detection Apparatus and Method." A true and correct copy of the '222 patent is attached hereto as Exhibit D.

30. Wistron and AOpen seek a judicial determination and declaration that no valid and enforceable claim of the '222 patent is infringed by Wistron and AOpen.

31. Wistron and AOpen seek a judicial determination and declaration that the '222 patent is invalid because it fails to satisfy the conditions and requirements for patentability as set forth, *inter alia*, in Sections 101, 102, 103, and/or 112 of Title 35 of the United States Code.

PRAYER FOR RELIEF

WHEREFORE, Wistron and AOpen pray for judgment as follows:

1. Declaring that no valid and enforceable claim of the patents-in-suit is infringed by Wistron and AOpen;

2. Declaring that Adams and its officers, employees, agents, alter egos, attorneys, and any persons in active concert or participation with them be restrained and enjoined from further prosecuting or instituting any action against Wistron and AOpen claiming that the patents-in-suit are valid, enforceable, or infringed, or from representing that the products or services of Wistron and AOpen infringe the patents-in-suit;

3. A judgment declaring this case exceptional under 35 U.S.C. § 285 and awarding Wistron and AOpen their attorneys' fees and costs in connection with this case; and

4. Awarding Wistron and AOpen such other and further relief as the Court deems just and proper.

Dated: October 1, 2010

K&L GATES LLP

By: [illegible]

Howard Chen (SBN 257393)
Harold H. Davis, Jr. (SBN 235552)
Rachel Davidson (SBN 215517)

Attorneys for Plaintiffs
WISTRON CORPORATION,
AOPEN INCORPORATED AND
AOPEN AMERICA INCORPORATED

Exhibit A

US005379414A

United States Patent [19]

Adams

[11] Patent Number: 5,379,414

[45] Date of Patent: Jan. 3, 1995

[54] **SYSTEMS AND METHODS FOR FDC ERROR DETECTION AND PREVENTION**

[76] Inventor: **Phillip M. Adams**, 1466 Chandler Dr., Salt Lake City, Utah 84103

[21] Appl. No.: **911,409**

[22] Filed: **Jul. 10, 1992**

[51] **Int. Cl.**^{6} **G06F 11/34**; H04L 1/18

[52] **U.S. Cl.** **395/575**; 371/62; 371/32; 371/33; 395/275

[58] **Field of Search** 371/62, 8.2, 9.1, 11.1, 371/11.3, 21.1, 28, 21.6, 37.1, 40.1, 68.1, 66, 32, 33, 34, 35, 48; 380/4, 24, 25; 395/425, 575, 400, 375, 550, 275

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,908,099	9/1975	Borbas et al.	179/175.2 R
4,942,606	7/1990	Kaiser et al.	380/4
4,996,690	2/1991	George et al.	371/37.1
5,093,910	3/1992	Tulpule et al.	395/575
5,212,795	5/1993	Hendry	395/725
5,233,692	8/1993	Gajjard et al.	395/325
5,237,567	8/1993	Nay et al.	370/85.1

OTHER PUBLICATIONS

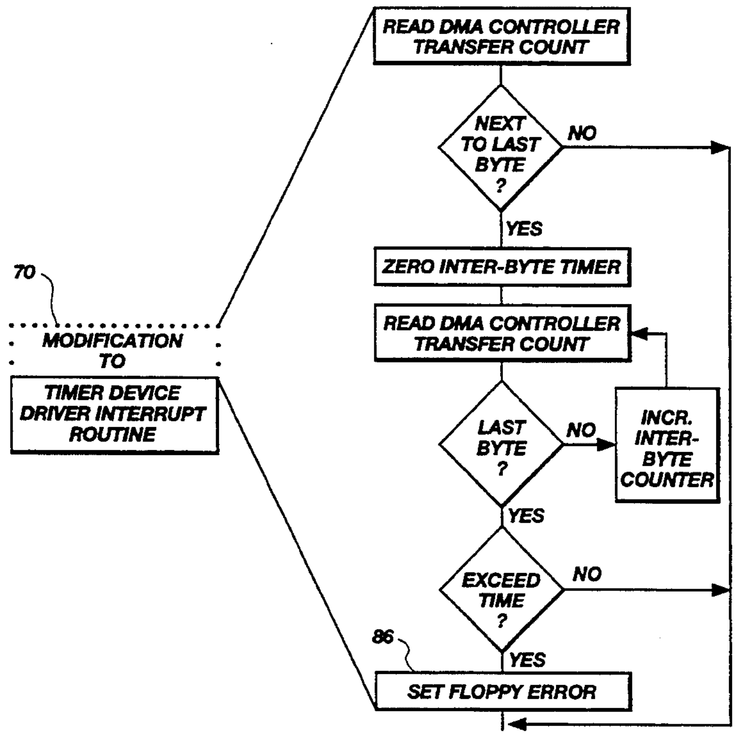
NEC Electronics Inc., "IBM-NEC Meeting for uPD765A/uPD72065 Problem" (U.S.A., May 1987).
Intel Corporation, Letter to customers from Jim Sleezer, Product Manager, regarding FDC error and possible solutions (U.S.A., May 2, 1988).
Adams, Phillip M., Nova University, Department of Computer Science, "Hardware-Induced Data Virus," Technical Report TR-881101-1 (U.S.A., Nov. 14, 1988).
Advanced Military Computing, "Hardware Virus Threatens Databases," vol. 4, No. 25, pp. 1 & 8 (U.S.A., Dec. 5, 1988).
Intel Corporation, "8237A/8237A-4/8237A-5 High Performance Programmable DMA Controller" (U.S.A., date unknown).
Intel Corporation, "8272A Single/Double Density Floppy Disk Controller" (U.S.A., Date unknown).

Primary Examiner—Robert W. Beausoliel, Jr.
Assistant Examiner—Dieu-Minh Le
Attorney, Agent, or Firm—Berne S. Broadbent; Gary D. E. Pierce

[57] **ABSTRACT**

A system and method which provides a complete software implementation of a device driver that is capable of detecting an undetectable data corruption problem without hardware redesign and/or internal modification to an existing FDC. The approach taken consists of software DMA shadowing and use of a software decoding network which allows the implementation of the invention to require a small amount of memory and only degrade the performance of the computer system a minimal amount when floppy diskette write operations occur.

7 Claims, 5 Drawing Sheets



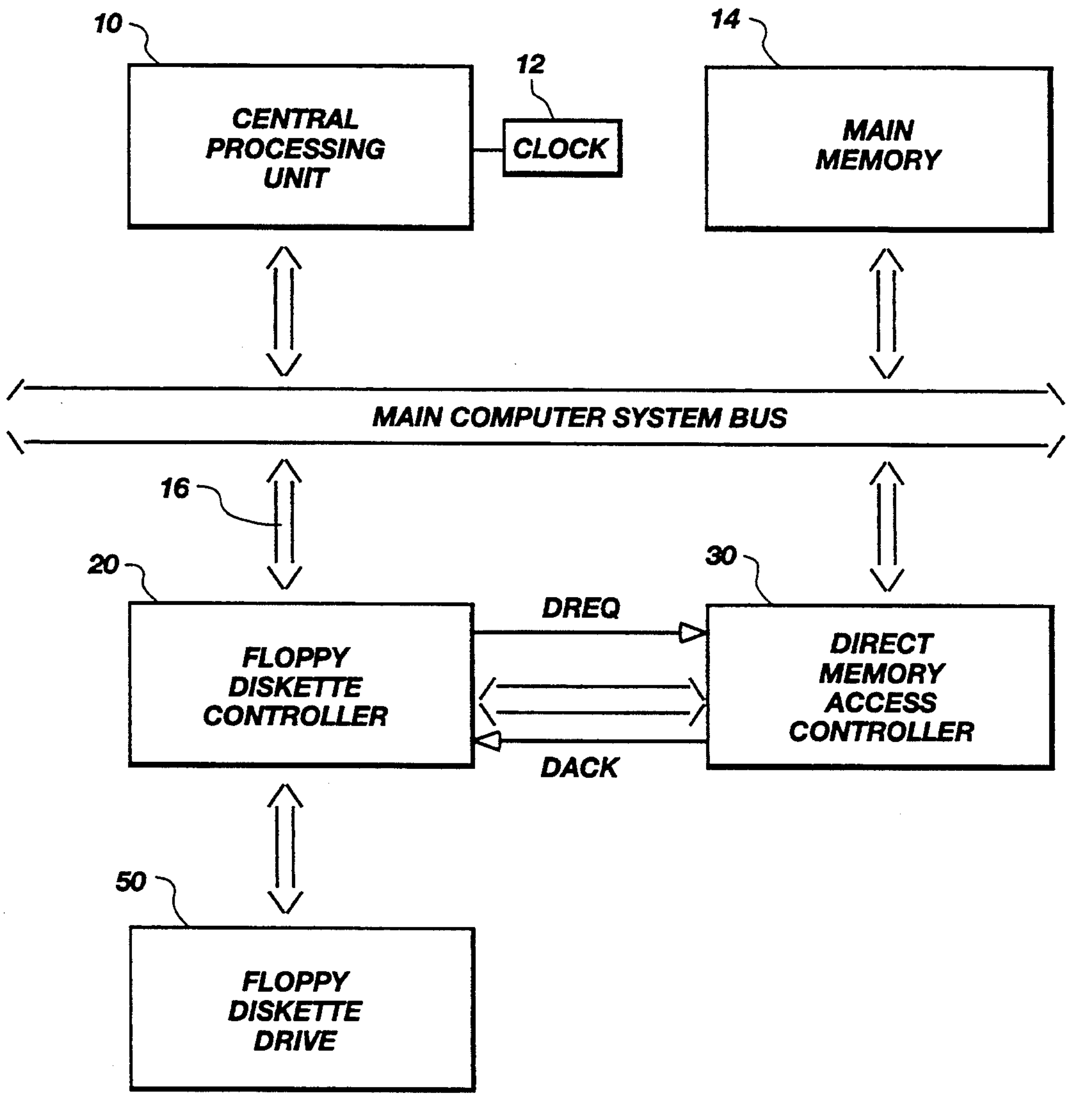

10
CENTRAL PROCESSING UNIT
12
CLOCK
14
MAIN MEMORY
MAIN COMPUTER SYSTEM BUS
16
20
FLOPPY DISKETTE CONTROLLER
DREQ
DACK
30
DIRECT MEMORY ACCESS CONTROLLER
50
FLOPPY DISKETTE DRIVE

Fig. 1

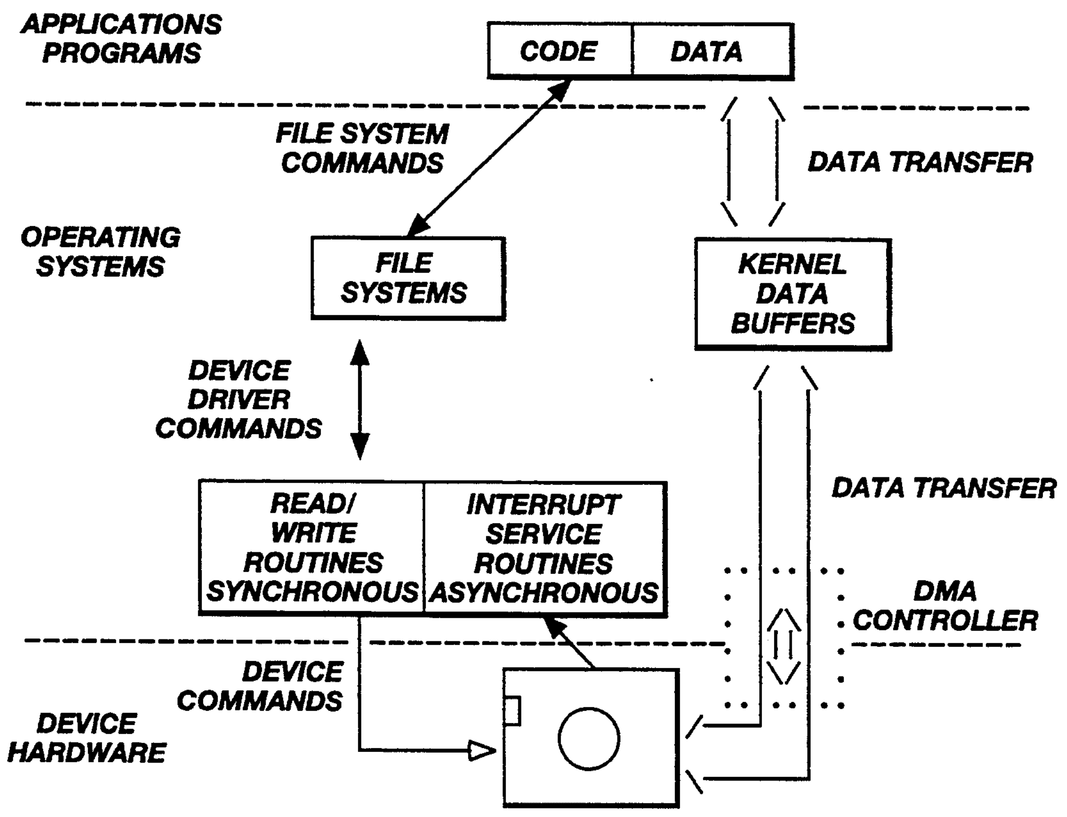

APPLICATIONS PROGRAMS
CODE
DATA
FILE SYSTEM COMMANDS
DATA TRANSFER
OPERATING SYSTEMS
FILE SYSTEMS
KERNEL DATA BUFFERS
DEVICE DRIVER COMMANDS
DATA TRANSFER
READ/ WRITE ROUTINES SYNCHRONOUS
INTERRUPT SERVICE ROUTINES ASYNCHRONOUS
DMA CONTROLLER
DEVICE COMMANDS
DEVICE HARDWARE

Fig. 2

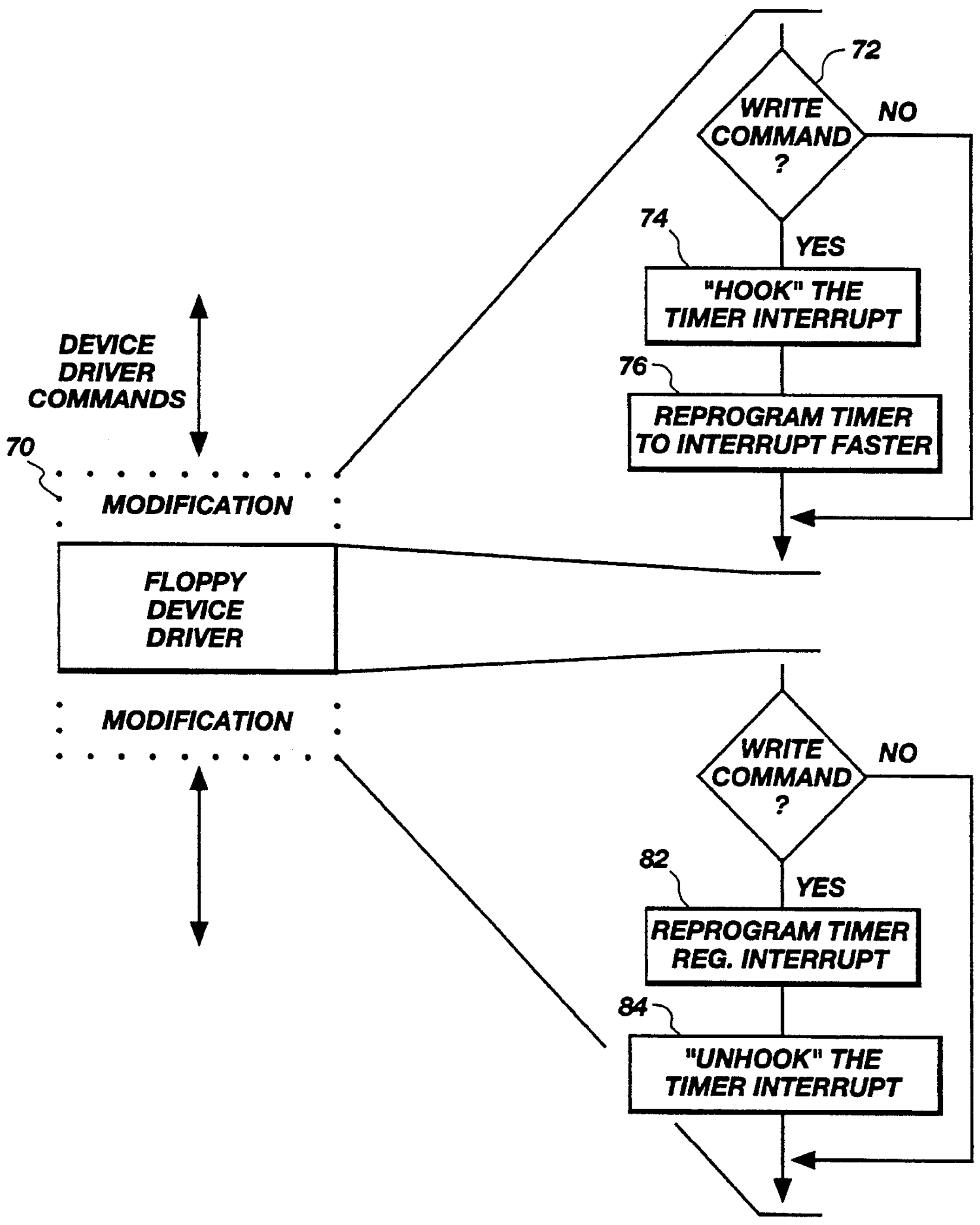

DEVICE DRIVER COMMANDS
70
MODIFICATION
FLOPPY DEVICE DRIVER
MODIFICATION
72
WRITE COMMAND ?
NO
74
YES
"HOOK" THE TIMER INTERRUPT
76
REPROGRAM TIMER TO INTERRUPT FASTER
WRITE COMMAND ?
NO
82
YES
REPROGRAM TIMER REG. INTERRUPT
84
"UNHOOK" THE TIMER INTERRUPT

Fig. 3
]

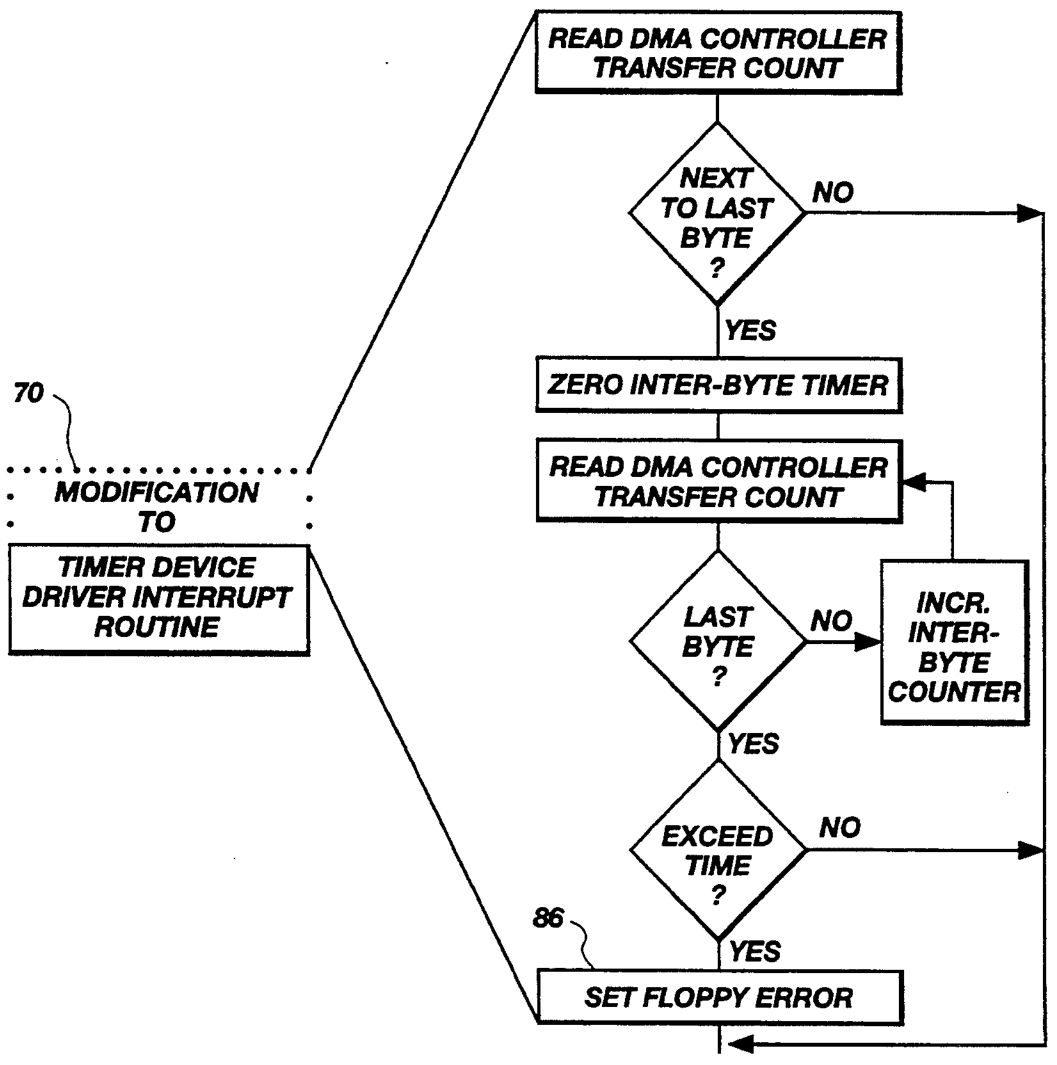

70
MODIFICATION TO
TIMER DEVICE DRIVER INTERRUPT ROUTINE
READ DMA CONTROLLER TRANSFER COUNT
NEXT TO LAST BYTE ?
NO
YES
ZERO INTER-BYTE TIMER
READ DMA CONTROLLER TRANSFER COUNT
LAST BYTE ?
NO
INCR. INTER-BYTE COUNTER
YES
EXCEED TIME ?
NO
YES
86
SET FLOPPY ERROR

Fig. 4

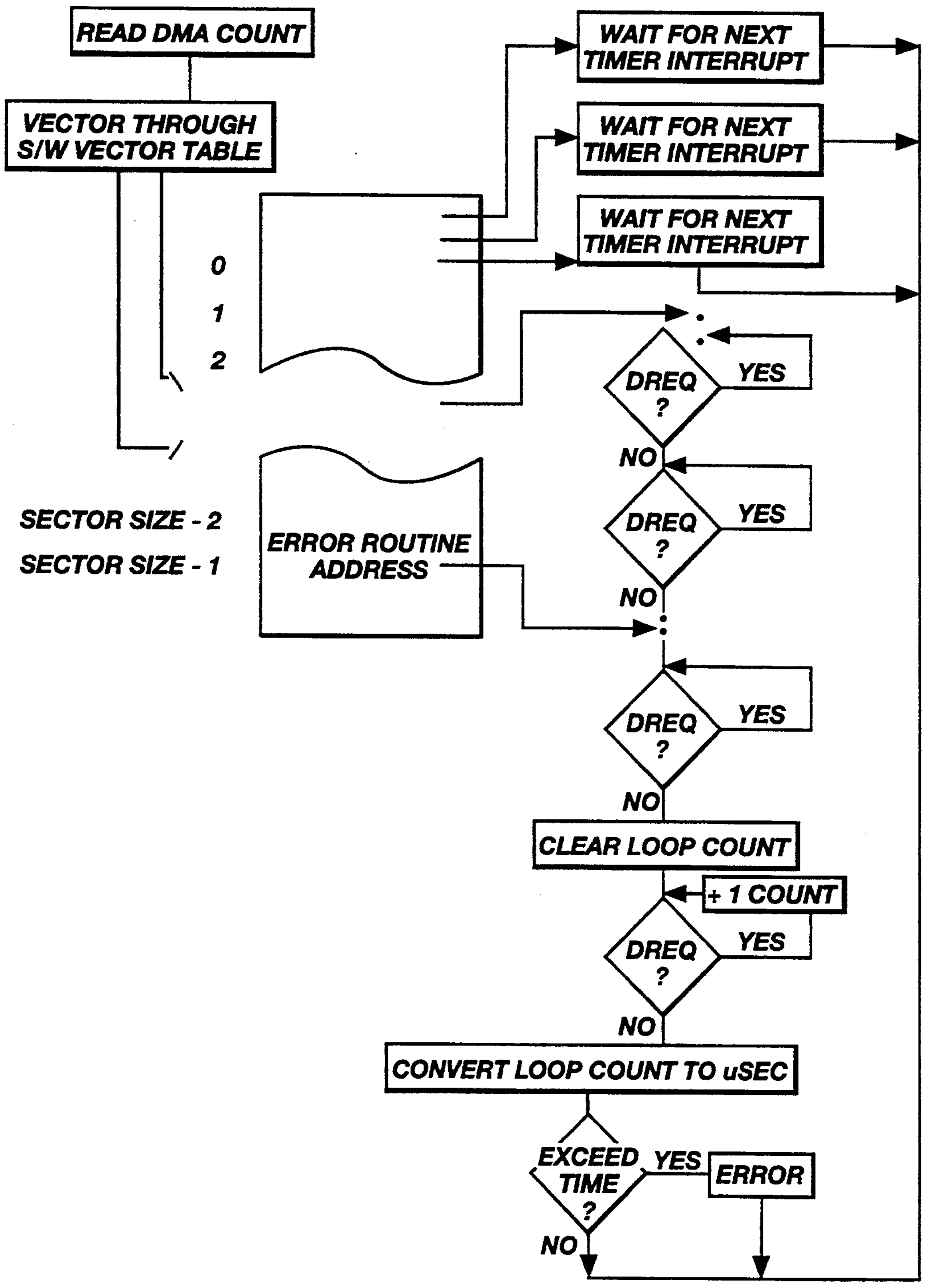

READ DMA COUNT
VECTOR THROUGH S/W VECTOR TABLE
0
1
2
SECTOR SIZE - 2
SECTOR SIZE - 1
ERROR ROUTINE ADDRESS
WAIT FOR NEXT TIMER INTERRUPT
WAIT FOR NEXT TIMER INTERRUPT
WAIT FOR NEXT TIMER INTERRUPT
DREQ ?
YES
NO
DREQ ?
YES
NO
DREQ ?
YES
NO
CLEAR LOOP COUNT
+ 1 COUNT
DREQ ?
YES
NO
CONVERT LOOP COUNT TO uSEC
EXCEED TIME ?
YES
ERROR
NO

Fig. 5

SYSTEMS AND METHODS FOR FDC ERROR DETECTION AND PREVENTION

BACKGROUND

1. The Field of the Invention

This invention relates to the detection and recovery procedure of an undetected Floppy Diskette Controller ("FDC") data error where data corruption occurs and, more particularly, to novel systems and methods implemented as a software-only device driver which eliminates the need for hardware redesign and/or fabrication of new FDCs.

2. The Background Art

Computers are now used to perform functions and maintain data which is critical to many organizations. Businesses use computers to maintain essential financial and other business data. Computers are also used by government to monitor, regulate, and even activate, national defense systems. Maintaining the integrity of the stored data is essential to the proper functioning of these computer systems, and data corruption can have serious (even life threatening) consequences.

Most of these computer systems include diskette drives for storing and receiving data on floppy diskettes. For example, an employee of a large financial institution might have a personal computer that is attached to the main system. In order to avoid processing delays on the mainframe, the employee may routinely transfer data files from the host system to his local personal computer and then back again, temporarily storing data on a local floppy diskette. Similarly, an employee with a personal computer at home may occasionally decide to take work home, transporting data away from and back to the office on a floppy diskette.

Data transfer to and from a floppy diskette is controlled by a device called a Floppy Diskette Controller ("FDC"). The FDC is responsible for interfacing the computer's Central Processing Unit ("CPU") with the physical diskette drive. Significantly, since the diskette is spinning, it is necessary for the FDC to provide data to the diskette drive at a specified data rate. Otherwise, the data will be written to the wrong location on the diskette.

The design of the FDC accounts for situations when the data rate is not adequate to support the rotating diskette. Whenever this situation occurs, the FDC aborts the operation and signals the CPU that a data underrun condition has occurred. Unfortunately, however, it has been found that a design flaw in many FDCs makes it impossible to detect all data underrun conditions. This flaw has, for example, been found in the NEC 765, INTEL 8272 and compatible Floppy Diskette Controllers. Specifically, data loss and/or data corruption can occur during data transfers to diskettes (or even tape drives and other media which employ the FDC), whenever the last data byte of a sector being transferred is delayed for more than a few microseconds. Furthermore, if the last byte of a sector write operation is delayed too long then the next (physically adjacent) sector of the diskette will be destroyed as well.

For example, it has been found that these FDCs cannot detect a data underrun on the last byte of a write operation to a sector of a diskette. Consequently, if the FDC is preempted during a data transfer (thereby delaying the transfer), and an underrun occurs on the last byte of a sector, the following occurs: (1) the underrun

flag does not get set, (2) the last byte written to the diskette is made equal to the previous byte written, and (3) CRC is generated on the altered data. The result is that incorrect data is written to the diskette and validated by the FDC.

Conditions under which this problem may occur can be identified by simply identifying those conditions that can delay data transfer to the diskette drive. In general, this requires that the computer system be engaged in "multi-tasking" operation or in overlapped input/output ("I/O") operation. Multi-tasking is the ability of a computer operating system to simulate the concurrent execution of multiple tasks. Importantly, concurrent execution is only "simulated" because there is only one CPU, and it can only process one task at a time. Therefore, a system interrupt is used to rapidly switch between the multiple tasks, giving the overall appearance of concurrent execution.

MS-DOS and PC-DOS, for example, are single-task operating systems. Therefore, one could argue that the problem described above would not occur. However, there are a number of standard MS-DOS and PC-DOS operating environments that simulate multi-tasking and are susceptible to the problem. The following environments, for example, have been found to be prime candidates for data loss and/or data corruption due to the FDC: local area networks, 327x host connections, high density diskettes, control print screen operations, terminate and stay resident (TSR) programs. The problem has also been found to occur as a result of virtually any interrupt service routine. Thus, unless the MS-DOS and PC-DOS operating systems disable all interrupts during diskette transfers, they are also susceptible to data loss and/or corruption.

Perhaps the best way to demonstrate the FDC error is to simulate a great deal of system activity. In other words, make the computer system act as though it were performing a large number of complex tasks all at one time. The problem has accordingly been demonstrated in systems using MS/PC-DOS operating systems by means of a simple test program. First, a clock program is executed and becomes a TSR task having the responsibility of servicing the timer interrupt (Ox1C) and updating the time on the screen. Second, a MS/PC-DOS diskette program is executed which writes a sector to the diskette using the BIOS interface interrupt (Ox13) and then reads the sector back. Once the sector has been written and read back the data is compared to determine whether or not an undetected error has occurred. A running total of both detected and undetected errors can then be output to the display. The results of using such a test program on various machines was quite astonishing. For example, the IBM PS/2 series seemed most susceptible to the problem, with roughly a 30% undetected error rate.

The UNIX operating system is a multi-tasking operating system, and it is extremely simple to create an environment that can cause the problem. One of the more simple examples is to begin a large transfer to the diskette and place that task in the background. After the transfer has begun then begin to display (cat) the contents of a very large file. The purpose of the video access is to force the video buffer memory refresh logic on DMA channel 1, along with the video memory access, to preempt the FDC operations occurring on DMA channel 2 (which is lower priority than channel 1). This example creates the classic overlapped I/O environment and can force the FDC into an undetectable error condition. More rigorous examples could include the concurrent transfer of data to or from a network or tape drive using a high priority Direct Memory Access ("DMA") channel while the diskette transfer is active. Clearly, the number of possible error producing examples is infinite and very possible in this environment.

For all practical purposes the OS/2 operating system can be regarded as a UNIX derivative. In other words, OS/2 suffers from the same problems that UNIX does. There are, however, two significant differences between OS/2 and UNIX. First, OS/2 semaphores video updates with diskette operations in an effort to avoid forcing the FDC problem to occur. However, any direct access to the video buffer, in either real or protected mode, during a diskette transfer will bypass this safe-guard and render OS/2 in the same condition as UNIX. Second, OS/2 incorporates a unique command that attempts to avoid the FDC problem by reading back every sector that is written in order to verify that the operation completed successfully. This command is an addition to the MODE command (MODE DSKT VER=ON). With these changes, data loss and/or data corruption should occur less frequently than before, but it is still possible for the FDC problem to destroy data that is not related to the current sector operation.

There are a host of other operating systems that are susceptible to the FDC problem just like DOS, OS/2, and UNIX. However, these systems may not have an install base as large as DOS, OS/2 or UNIX, and there may, therefore, be little emphasis on addressing the problem. Significantly, as long as the operating system utilizes the FDC and services system interrupts, the problem can manifest itself. This can, of course, occur in computer systems which use virtually any operating system.

Some in the computer industry have suggested that the FDC problem is extremely rare and difficult to reproduce. Admittedly, the problem is often very difficult to detect during normal operation because of its random characteristics. The only way to visibly detect this problem is to have the FDC corrupt data that is critical to the operation at hand. There may, however, be many locations on the diskette that have been corrupted, but not accessed. Studies have recently demonstrated that the FDC problem is quite easy to produce and may be more common than heretofore believed.

Computer users may, in fact, experience this problem frequently and not even known about it. After formatting a diskette, for example, the system may inform the user that the diskette is bad, although the user finds that if the operation is performed again on the same diskette everything is fine. Similarly, a copied file may be unusable, and the computer user concludes that he or she just did something wrong. For many in this high-tech world, it is very difficult to believe that the machine is in error and not ourselves. It remains a fact, however, that full diskette back-ups are seldom restored, that all instructions in programs are seldom, if ever, executed, that diskette files seldom utilize all of the allocated space, and that less complex systems are less likely to exhibit the problem.

Additionally, the first of these FDCs were shipped over 10 years ago. The devices were primarily used at that time in special-purpose operations in which the FDC problem would not normally be manifest. Today, on the other hand, the FDCs are incorporated into

general-purpose computer systems that are capable of concurrent operation (multi-tasking or overlapped I/O). Thus, it is within today's environments that the problem is most likely to occur by having one of the operations delay the data transfer to the diskette. The more complex the computer system, the more likely it is to have one activity delay another, thereby creating the FDC error condition.

In short, the potential for data loss and/or data corruption is present in all computer systems that utilize this type of FDC, presently estimated at about 25 million personal computers. The design flaw in the FDC causes data corruption to occur and manifest itself in the same manner as a destructive computer virus. Furthermore, because of its nature, this problem has the potential of rendering even secure databases absolutely useless.

Those skilled in the art have suggested various ways of addressing the FDC problem. Unfortunately, however, each of these prior solutions has significant associated costs, risks and/or disadvantages.

For example, perhaps the most desirable solution is to have the manufacturer of the FDC provide a new FDC that alleviates the problem. This approach is, however, only a partial solution since many of the current systems have the FDC soldered into a circuit board. It would, of course, entail significant effort and/or cost to remove the current FDC and replace it with a new one.

Add-on hardware devices have similarly been suggested which could detect the FDC error condition and force it to be acknowledged by the CPU. Like a new FDC, however, such devices are at best inconvenient to install and use and are thus unlikely to be used by many computer users.

In an effort to avoid the disadvantages of a hardware solution, some read back and verify programs, like the IBM OS/2 MODE command, have been developed and installed. Such programs typically require that the FDC device driver perform single-sector writes, read the previously written sector back into a sector buffer in the FDC device driver, and then compare the data that was supposed to be written to the floppy with the data contained in the readback buffer. This process is performed until all data compares properly.

There are a number of problems that occur when employing this detection and a recovery procedure. Three of the most important problems are: (1) the size of the FDC device driver grows due to sector readback buffers required; (2) unacceptable performance is encountered because each sector must be written, the diskette must then make a full revolution for the sector to be readback, and finally the readback buffer must be compared with the original data to determine the success or failure of the I/O operation (thus causing all diskette transfers to execute at roughly one-third their normal speed); and (3) this approach is only partially effective in eliminating the FDC problem since it does not account for the data corruption that can occur to the physically adjacent sector when data transfer is significantly delayed. In short, the write/read/compare approach does not adequately protect the data from being corrupted, it causes more memory to be utilized by the operating system, and it degrades performance of the floppy diskette to an intolerable level. As a result, this approach has likewise not generally been adopted.

BRIEF SUMMARY AND OBJECTS OF THE INVENTION

In view of the foregoing, it is a primary object of the present invention to provide a system and method for the detection and recovery procedure of an undetected FDC data error where data corruption occurs.

It is also an object of the present invention to provide a software-only device driver which eliminates the need for hardware redesign and/or fabrication of new FDCs.

In addition, it is an object of the present invention to provide a solution to an I/O controller's (FDC) defect using DMA shadowing.

It is a further object of the present invention to provide a system and method for the detection and recovery procedure of an undetected FDC data error which reduces system performance only a minimal amount during floppy write operations.

It is a still further object of the present invention to provide a solution to an I/O controller's (FDC) defect using variable processor speed DMA shadowing via a software decoding network.

Consistent with the foregoing objects, and in accordance with the invention as embodied and broadly described herein, a system and method are disclosed in one embodiment of the present invention as including a device driver that is capable of detecting an undetectable data corruption problem without hardware redesign and/or internal modification to an existing FDC. The approach taken consists of DMA shadowing and use of a software decoding network which allows the implementation of the invention to require a small amount of memory and only degrade the performance of the computer system a minimal amount when floppy diskette write operations occur.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects and features of the present invention will become more fully apparent from the following description and appended claims, taken in conjunction with the accompanying drawings. Understanding that these drawings depict only typical embodiments of the invention and are, therefore, not to be considered limiting of its scope, the invention will be described with additional specificity and detail through use of the accompanying drawings in which:

FIG. **1** is simplified block diagram which illustrates the architecture of most computer systems employing an floppy diskette controller (FDC);

FIG. **2** is a block diagram which illustrates the typical association between application programs, operating systems, device drivers and computer system hardware (in this example, a floppy diskette);

FIG. **3** is a flow chart depicting one presently preferred embodiment of the modifications that are applied to the diskette device driver in order to allow the error detection/prevention system and method of the present invention to be activated;

FIG. **4** is a flow chart depicting one presently preferred embodiment of the modifications that are made to the timer Interrupt Service Routine (ISR) so as to allow timing of the last byte's DREQ/DACK cycle in accordance with the present invention; and

FIG. **5** is a flow chart depicting one presently preferred embodiment of a software decoding network (software vector-table) for use in connection with the error detection/prevention system and method of the present invention, the network having one code point/-entry for each possible transfer byte in the sector.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

It will be readily understood that the components of the present invention, as generally described and illustrated in the Figures herein, could be arranged and designed in a wide variety of different configurations. Thus, the following more detailed description of the embodiments of the system and method of the present invention, as represented in FIGS. **1** through **5**, is not intended to limit the scope of the invention, as claimed, but it is merely representative of the presently preferred embodiments of the invention.

The presently preferred embodiments of the invention will be best understood by reference to the drawings, wherein like parts are designated by like numerals throughout.

The architecture of a typical computer system is illustrated in FIG. **1**. The computer system's Central Processing Unit ("CPU") **10** and main memory **14** are located inside the system unit. The instructions and data used by the CPU **10** are kept in main memory **14** during computer work sessions. Main memory **14** is, however, not a permanent storage place for information; it is active only when the computer system is on. Thus, to avoid losing the data, it must be saved on some type of storage device. For example, the computer system may use a "hard disk" storage device which is permanently installed in the computer system. Most computer systems have at least one floppy diskette drive **50** that receives a removable floppy diskette. The floppy diskette likewise is used for "permanent" storage of data or software outside of the computer system and is especially useful for transferring data and information between separate computer systems.

In transferring data to a floppy diskette, the CPU **10** typically programs the Direct Memory Access ("DMA") controller **30** for an input/output ("I/O") transfer, issues a command to the Floppy Diskette Controller ("FDC") **20** to begin the I/O transfer, and then waits for the FDC to interrupt with a completion interrupt signal. It is also possible to perform Programmed I/O ("PIO") directly between the CPU **10** and the FDC **20** without involving the DMA controller **30**, as illustrated by the broken arrows **16**. This latter approach is seldom used; the majority of computer systems employ DMA I/O transfers to and from the floppy diskette. The invention will thus be described below with particular reference to the DMA controller **30**. If PIO is employed, however, then the I/O transfer is totally controlled by the CPU **10** because the CPU is required to pass each and every data byte to the FDC **20**. As a result, the "DMA shadowing" system and method of this invention can be directly applied to the PIO data stream because the CPU **10** already is controlling the I/O transfer, as will become more readily apparent from the discussion which follows.

Virtually all computer systems must have a system clock **12**. The system clock is necessary when initiating an I/O transfer to the diskette drive **50** because one must not only control the data transfer, but also the drive motor. In this regard, it is important to know when the diskette drive motor has brought the diskette spin rate up to the nominal RPM required for the data transfer to be successful.

As an example, in IBM Personal Computers and compatibles, the system clock 12 interrupts the CPU 10 18.6 times per second (roughly once every 54 milliseconds). This interrupt is used to determine such things as diskette drive motor start and stop time. There are also a host of other time-dependent operations in the computer system that require this granularity of timing.

The typical association between application programs, operating systems, device drivers and hardware is depicted in FIG. 2. The example presented is the floppy diskette.

As illustrated in FIGS. 3 and 4, the system and method of the present invention includes an interposer routine 70 which is placed between the application's request for floppy service and the current floppy device driver. The interposer routine 70 is actually a new or modified device driver that implements the detection and recovery for the undetected FDC data corruption. As shown, the interposer 70 first determines if the operation is a floppy diskette write operation (72). If so, the major function of interposer 70 is to insert itself between the application request for floppy service and the floppy device driver that will service the request. In the PC/MS-DOS environment, this can be accomplished by "hooking" the INT Ox13 interrupt vector (74) and directing it to the interposer routine and then reprogramming the timer to interrupt faster than normal (76) (e.g., every 4–7 milliseconds).

As will become more fully apparent from the discussion which follows, once a floppy write operation is detected, a software decoding network call vector is preferably installed (see FIG. 5), the current byte count is read, and DMA shadowing begins. If the time is too long, an error condition is forced (86). The system clock 12 is then reprogrammed (82) to interrupt normally (e.g., every 54 milliseconds), and timer interrupt is "unhooked" (84) until the next floppy write operation. Clearly, one could allow the timer to always interrupt at the accelerated rate and then check in the timer Interrupt Service Routine ("ISR") if a diskette write operation is active, but this approach is not as performance-minded as the one presented herein.

As used herein, "DMA shadowing" means monitoring byte transfers and then timing the last byte of a sector's DREQ to DACK signals. Importantly, there are, of course, a number of ways of determining when the DREQ is present and when the DACK is present. The sample code set forth below is only one approach. The present invention includes the use of any "DMA shadowing" whether the DREQ and DACK signals are detected at the DMA controller, CPU, system bus or FDC. This includes both explicit means, as presented in the sample code, and implicit means, such as inferring

the state of the DREQ/DACK cycle from various components in the system that are triggered or reset from such signal transitions. A typical example is that the DACK can cause the Terminal Count (TC) signal to be asserted. Therefore, one can imply from the detection of TC that DACK has occurred.

In other words, whenever an application requests a floppy write operation, the system clock 12 is reprogrammed to interrupt every 4 to 7 milliseconds. Each time the system clock interrupts, the current byte count in the DMA controller transfer register (countdown register) is read. Once the byte counter has reached the last byte, the signal transition from DREQ to DACK is timed. If the time is greater than the time that will insure data integrity, an error condition is forced which is similar to the one the FDC hardware would produce if not defective. Finally, to avoid corruption to the adjacent sector the system causes the FDC to abort an operation if the timing between the DREQ and the DACK extends past the corruption point, but not to the adjacent sector destruction point, and return an error condition as previously described.

In order for the system to maintain proper operation, it is necessary that the interposer 70 save the original INT Ox13 contents (address of the original INT Ox13 Interrupt Service Routine) and invoke the original when necessary. Additional aspects of the interposer function are discussed below in connection with the other features of the device driver.

The following code fragment from the actual device driver demonstrates the interposer operation. This code fragment, together with the other sample code set forth herein, illustrates in more detail one preferred embodiment of a software routine derived from the block diagram of FIGS. 3 through 5. Those of ordinary skill in the art will, of course, appreciate that various modifications to the specific sample code may easily be made without departing from the essential characteristics of the invention. Thus, the illustrative code, and the accompanying description, is intended only as an example, and it simply illustrates one presently preferred embodiment that is consistent with the invention as claimed herein.

The examples set forth below relate to an implementation of the system and method of the present invention for use on an IBM Personal Computer running the PC/MS-DOS operating system. Similar versions have, however, been developed to operate in the UNIX and OS/2 environments. The invention is not limited to use with any particular operating system, and adaptations and changes which may be required for use with other operating systems will be readily apparent to those of ordinary skill in the art.

```
PROCEDURE:   _INT13_isr
REMARKS:     _INT13_isr is responsible for receiving the INT 13 diskette interrupts from
             the O/S (BIOS). A check is made to see if the requested operation is a
             WRITE and the drive is a diskette (0 or 1). If so, then the timer is enabled
             and the system is set-up to delay the last (512th) byte of the transfer to
             generate an undetected underrun/overrun condition.
_INT13_isr proc far
            mov   _TEXT : active, 0          ; Clear Diskette Write Active Flag
            mov   _TEXT : errors, 0          ; Clear Diskette Error Flag
            cmp   dl,2                       ; Check Drive Number for Diskette 0 or 1
            jb    chk_write                  ; If Diskette, Then Check for Write
            jmp   _TEXT : int13_ptr          ; Otherwise, enter Original INT 13
chk_write:
            cmp   ah,3                       ; Check For Diskette Write Operation
            je    set_active                 ; If Write, Then Diskette Write Active
            jmp   _TEXT : int13_ptr          ; Otherwise, Enter Original INT 13
```

-continued

```
set_active:
            push  ds                          ; Save DS Register
            mov   _TEXT : sectors,al          ; Save Number of Sectors to Transfer
            mov   _TEXT : track,ch            ; Save Track Number
            mov   _TEXT : sector,cl           ; Save Sector Number
            mov   _TEXT : head, dh            ; Save Head Number
            mov   _TEXT : drive,dl            ; Save Drive Number
            mov   _TEXT : active,OFFH         ; Set Diskette Write Active Flag
            call  NEAR PTR_Timer_enable       ; Enable Timer for "Shadowing" DMA
            pop   ds                          ; Restore DS Register
            pushf                             ; Simulate An Interrupt
            call  _TEXT : int13_ptr           ; Enter Original INT 13 ISR
            pushf                             ; Save INT 13 Flags
            call  NEAR PTR_Timer_disable      ; Disable the Timer "Shadow" Routine
            mov   _TEXT : active,0            ; Clear Diskette Write Active Flag
            popf                              ; Restore INT 13 Flags
            cmp   _TEXT : errors,0            ; Were Errors Detected by Timer ISR
            jz    _INT13_exit                 ; If Zero, Then no Errors-Exit
            stc                               ; Otherwise, Set Error Indicator
            mov   _TEXT : errors,0            ; And Clear Error Flag
_INT13_exit:
            ret   2                           ; Eliminate Entry Flags
_INT13_isr endp
```

The foregoing interposer routine checks to see if the request is a write operation. If so, then it calls _Timer_enable (reprogram the system clock), calls the original INT Ox13 Interrupt Service Routine (perform the actual write operation while DMA Shadowing is enabled), and finally calls _Timer_disable (reprograms the system clock to the original clock interrupt rate of approximately 54 milliseconds).

As mentioned above, the interposer invokes system clock management routines (_Timer_enable and _Timer_disable). These two functions provide the system clock to interrupt at an accelerated rate that is 8 to 10 times faster than normal. The disable routine (see FIG. 3) returns the system clock interrupt rate to the normal interrupt rate. Thus, they are inverse identities which make the system behave normally at all time without significant performance degradation (<10%) and only during floppy write operations. At any other time the performance is virtually unchanged.

The following code fragment depicts the operation of the _Timer_enable function (76):

```
PROCEDURE:   _TIMER_enable
REMARKS:     Timer enable is responsible for 1) moving INT 8 vector to INT 0 × 60, 2) Initializing
             INT 8 vector to _Timer_irs, and 3) Reprogramming the 8253 to
             interrupt 128 times faster. (Clock count * 0.840 Microseconds)
_Timer_enable proc  near
         pushf                                  ; Save Current Flags Register
         cli                                    ; Disable Interrupts
         push     ax                            ; Save AX Register
         push     bx                            ; Save BX Register
         push     es                            ; Save ES Register
         xor      ax,ax                         ; Zero AX Register
         mov      es,ax                         ; Set ES to Absolute Zero Segment
         mov      ax,es:[8*4]                   ; Obtain Offset of INT 8
         mov      bx,es:[(B*4)+2]               ; Obtain Segment of INT 8
         mov      _TEXT : int8_off,ax           ; Move INT 8 Offset to INT 8 Offset
         mov      _TEXT : int8_seg,bx           ; Move INT 8 Segment to INT 8 Segment
         mov      ax,es:[1CH*4]                 ; Obtain Offset of INT 1C
         mov      bx,es:[(1CH*4)+2]             ; Obtain Segment of INT 1C
         mov      _TEXT : int1C_off,ax          ; Move INT 1C Offset to INT IC Offset
         mov      _TEXT : int1C_seg,bx          ; Move INT 1C Segment to INT 1C Segment
         mov      _TEXT : ticks,0               ; Zero ticks for Use by New ISR
         in       al,020H                       ; Obtain 8259 Interrupt Mask Register
         push     ax                            ; Save 8259 IMR for Restoration
         mov      al,OFFH                       ; Mask All External Interrupts At 8
         out      021H,al                       ; Set-up New 8259 IMR
         mov      al,36H                        ; Establish Operational Mode of 8253
         out      43H,al                        ; Program Operation of 8253
         jmp      $+2                           ; Allow Time for I/O to Complete Op
         mov      al,0                          ; Load LSB of Down Counter
         out      40H,al                        ; Send to 8253
         jmp      $+2                           ; Allow Time for I/O to Complete Op
         mov      al,02H                        ; Load MSB of Down Counter
         out      40H,al                        ; Send to 8253
         jmp      $+2                           ; Allow Time for I/O to Complete Op
         mov      ax,OFFSET _TEXT : _Timer_isr  ; Obtain the Offset of _Timer _isr
         mov      bx,cs                         ; Obtain the Segment of _Timer_isr
         mov      es:[8*4], ax                  ; Install new INT 8 Offset
         mov      es:[(8*4)+2],bx               ; Install new INT 8 Segment
         mov      ax,OFFSET_TEXT : _INT1C_isr   ; Obtain the Offset of _INT_1C
         mov      bx,cs                         ; Obtain the Segment of _INT_1C
         mov      es: [1CH*4]ax                 ; Install new INT 1C Offset
;        mov      es:[(1CH*4)+2],bx             ; Install new INT 1C Segment
```

-continued

```
;         pop     ax                                ; Obtain old 8259 IMR from Stack
;         out     021H,al                           ; Set-up old 8259 IMR
Enable_exit:
          pop     es                                ; Restore ES Register
          pop     bx                                ; Restore BX Register
          pop     ax                                ; Restore AX Register
          popf                                      ; Restore Flags Register
          ret                                       ; Return to Caller
_Timer_enable endp
```

The foregoing _Timer_enable routine performs two major functions. First, this function "hooks" the interrupt vectors that are associated with system clock functions (INT Ox08 and INT Ox1C). The original values of these two interrupt vectors are saved for use in the newly installed interrupt service routines for these interrupts (_Timer_isr and _INT1C_isr). Next, the system cloak is reprogrammed to interrupt at the accelerated rate. However, now the system clock interrupts will be processed by the newly installed system clock interrupt service routines.

The following code fragment depicts the operation of the _Timer-disable function (82):

```
PROCEDURE:  _Timer_disable
REMARKS:    _Timer_disable is responsible for 1) Reprogramming the 8253 to
            interrupt 128 times slower and 2) Restoring the old INT 8
            interrupt vector from INT 60.
_Timer_disable proc   near
            pushf                           ; Save Current Flags Register
            cli                             ; Disable Interrupts
            push   ax                       ; Save AX Register
            push   bx                       ; Save BX Register
            push   es                       ; Save ES Register
;           in     al,020H                  ; Obtain 8259 Interrupt Mask Register
;           push   ax                       ; Save 8259 IMR For Restoration
;           mov    al,OFFH                  ; Mask All External Interrupts At 8
;           out    021H,al                  ; Set-up new 8259 IMR
            mov    al,36H                   ; Establish Operational Mode of 8253
            out    43H,al                   ; Program Operation of 8253
            jmp    $+2                      ; Allow Time for I/O to Complete Op
            mov    al,0                     ; Load LSB of Down Counter
            out    40H,al                   ; Send LSB to 8253
            jmp    $+2                      ; Allow time for I/O to Complete Op
            out    40H,al                   ; Send MSB to 8253
            jmp    $+2                      ; Allow time for I/O to Complete Op
            xor    ax,ax                    ; Zero AX Register
            mov    es,ax                    ; Set ES to Absolute Zero Segment
            mov    ax,_TEXT : int8_off      ; Obtain Offset of Original INT 8
            mov    bx,_TEXT : int8_seg      ; Obtain Segment of Original INT 8
            mov    es:[8*4],ax              ; Install Old INT 8 Offset
            mov    es:[(8*4) +2], bx        ; Install Old INT 8 Segment
            mov    ax,_TEXT : int1C_off     ; Obtain Offset of Original INT 1C
            mov    bx,_TEXT : int1C_seg     ; Obtain Segment of Original INT 1C
            mov    es:[1CH*4],ax            ; Install Old INT 1C Offset
            mov    es:[(1CH*4)+2],bx        ; Install Old INT 1C Segment
;           pop ax                          ; Obtain Old 8259 IMR From Stack
;           out    021H,al                  ; Set-up Old 8259 IMR
            pop    es                       ; Restore ES Register
            pop    bx                       ; Restore BX Register
            pop    ax                       ; Restore AX Register
            popf                            ; Restore Flags Register
            ret                             ; Return to Caller
_Timer_disable      endp
```

The _Timer-disable function above is responsible for restoring the system to its original state. In other words, the INT Ox08 and INT Ox1C interrupts are restored to their original values and the system clock is reprogrammed to interrupt at its original frequency.

As depicted graphically in FIG. 4 and described further below, a _Timer-isr routine is used for servicing the accelerated interrupt rate of the system clock. The reason that the system clock interrupt rate is accelerated is that during a normal 512 byte data transfer (the typical sector size) 16 microseconds are required for each data byte to be transferred to the FDC (High Density Diskette Mode). Therefore, a typical Sector transfer requires 512 times 16 microseconds, or 8,192 microseconds. If the diskette is a low density diskette then the sector transfer time is doubled to 16,384 microseconds (512 times 32 microseconds) because the FDC has half of the amount of data to store in the same rotational time frame (typically 360 RPM).

The following code fragment illustrates the _Timer_isr routine:

```
PROCEDURE:  _Timer_isr
REMARKS:    _Timer_isr is responsible for receiving the INT 8 timer interrupts from the 8253.
            Since the 8253 has been programmed to interrupt approximately 128 times
```

-continued

```
                    faster _Timer_isr must call the old INT 8 ISR routine every 128th interrupt.
                    _Timer_isr is also responsible for initiating a DMA
                    request on every timer interrupt.
_Timer_isr proc
          inc       _TEXT : ticks                 ; Increment the Interrupt Tick Counter
          cmp       _TEXT : active, OFFH          ; Check to see if Diskette Write Op
          jz        begin_isr                     ; If so, Then Begin Timer ISR
          jmp       chk_old                       ; Otherwise, Check if Old ISR Runs
begin_isr:
          cld                                     ; Clear the Direction Flag
          mov       _TEXT : AX_reg,ax             ; Save AX Register
          mov       _TEXT : BX_reg,bx             ; Save BX Register
          mov       _TEXT : CX_reg,cx             ; Save CX Register
          mov       _TEXT : DX_reg,dx             ; Save DX Register
          mov       _TEXT : SI_reg,si             ; Save SI Register
          out       OCH,al                        ; Clear the Byte Pointer Flip/Flop
          cmp       _TEXT : CHN1_flag,0           ; Should Channel 1 Be Masked (Diabl)
          jz        bypass_CHN1                   ; If Zero, Then Don't Mask Channel
mask_CHN1:
          mov       al,005H                       ; DMA Mask Register Value
          out       00AH,al                       ; Disable DMA Channels (1 for Now)
          jmp       $+2                           ; (Can't Disable Refresh on XT - CH
bypass_CHN1:
;         in        al,020H                       ; Obtain 8259 Interrupt Mask Register
;         push      ax                            ; Save 8259 IMR for Restoration
;         mov       al,OBDH                       ; Disable all but KBD and Diskette
;         out       021H,al                       ; Set-up New 8259 IMR
          in        al,08H                        ; Obtain 8237 DMA Status
          test      al,04H                        ; Check for Channel 2 DMA TC
          je        chk_change                    ; If no TC, Then Check for DMA Req
          jmp       exiting                       ; Otherwise, Exit Routine
chk_change:
          in        al,05H
          mov       dl,al
          in        al,05H
          mov       dh,al
          mov       cx,_TEXT : ratio
chk_change_A:
          in        al,05H
          mov       bl,al
          in        al,05H
          mov       bh,al
          cmp       bx,dx
          jne       chk_count
          in        al,05H
          mov       bl,al
          in        al,05H
          mov       bh,al
          cmp       bx,dx
          jne       chk_count
          loop      chk_change_A                  ; Loop if Processor is Fast
          jmp       exiting                       ; If not DRQ, Exit Routine
chk_count:
          mov       si,bx                         ; Save Current DMA Count
          and       si,01FFH                      ; Isolate Lower Sector Count
          shl       si,1                          ; Word Normalize SI Value
          jmp       _TEXT : DMA_Count_TBL[SI]     ; Shadow the DMA Transfers Via S/W
```

The foregoing _Timer_isr routine performs some sanity checks on the system to determine if the system is actually transferring data to the FDC. If a sector transfer is not in progress then _Timer_isr exits immediately. However, if a sector transfer is in progress then _Timer_isr obtains the remaining byte count of the sector transfer and vectors (jumps) through the software decoding network (DMA_Count_TBL) to the appropriate processing routine.

Although the system and method depicted in FIG. 4, could be implemented as is, it would require the timer to interrupt every 8, 16, or 32 microseconds. This level of interrupts would totally consume a PC's processing power, and on a PC/XT would/could not be sustained. Thus, in order to perform DMA shadowing without affecting the total system performance it is necessary to allow normal operations to continue as usual, but have an interrupt (the system clock) that will interrupt close to the end of the sector transfer so that the DREQ to DACK timing can be determined on the last byte of the sector transfer.

Clearly, it is possible to DMA shadow all 512 bytes during a sector transfer, but that would cause the CPU to be totally consumed during the entire sector transfer time. In other words, the potential of losing processing activities somewhere else in the system are greatly increased, such as in the case of serial communications. Therefore, the following clock interrupt strategy was developed to reduce the CPU involvement to a bare minimum during the floppy write operations with DMA Shadowing. Significantly, the timing strategy can be adjusted to any number of bytes of the sector transfer, from a few bytes to the entire sector count.

As indicated above, the last operation performed in the _Timer_isr routine is to vector through the software decoding network to the appropriate processing routine. This process is illustrated graphically in FIG. 5. The software decoding network (software vector-table) has one code point/entry for each possible transfer byte in the sector. The timer interrupt rate can now be in terms of 10's or 100's of byte transfer times because the vector table will cause the program execution to enter a cascade of DREQ/DACK checks only when it knows that the transfer (sector) will be complete prior to another timer interrupt. In short, the first X entries in the vector table will simply return (knowing that another timer interrupt will occur before the sector transfer completes). The latter Y entries will cascade from one DREQ/DACK detection to another (shadowing the DMA transfers) until the last byte is transferred. On the last byte being transferred, an instruction loop is counted and finally converted into microseconds to determine if the error has occurred.

This process is the fastest known technique for decoding and executing time-dependent situations. Memory space (the software decoding network vector table) is traded for processing time (the amount of time it would take for one routine to subsume all functionality encoded in each of the routines vectored to through the software decoding network vector table).

The following code fragment depicts a portion of the software decoding network employed in the DMA shadowing process of the presently preferred embodiment of the system and method of the present invention:

```
DMA_Count_TBL
          LABLE     WORD
          DW        OFFSET DMA_0       ; Entry for DMA Count 0
          DW        OFFSET DMA_1       ; Entry for DMA Count 1
          DW        OFFSET DMA_2       ; Entry for DMA Count 2
          DW        OFFSET DMA_3       ; Entry for DMA Count 3
          DW        OFFSET DMA_4       ; Entry for DMA Count 4
          DW        OFFSET DMA_5       ; Entry for DMA Count 5
          DW        OFFSET DMA_6       ; Entry for DMA Count 6
          DW        OFFSET DMA_7       ; Entry for DMA Count 7
          DW        OFFSET DMA_8       ; Entry for DMA Count 8
          DW        OFFSET DMA_9       ; Entry for DMA Count 9
          DW        OFFSET DMA_a       ; Entry for DMA Count a
          DW        OFFSET DMA_b       ; Entry for DMA Count b
          DW        OFFSET DMA_c       ; Entry for DMA Count c
          DW        OFFSET DMA_d       ; Entry for DMA Count d
          DW        OFFSET DMA_e       ; Entry for DMA Count e
          DW        OFFSET DMA_f       ; Entry for DMA Count f
          DW        OFFSET DMA_10      ; Entry for DMA Count 10
          DW        OFFSET DMA_11      ; Entry for DMA Count 11
          DW        OFFSET DMA_12      ; Entry for DMA Count 12
          DW        OFFSET DMA_13      ; Entry for DMA Count 13
          DW        OFFSET DMA_14      ; Entry for DMA Count 14
          DW        OFFSET DMA_15      ; Entry for DMA Count 15
          DW        OFFSET DMA_16      ; Entry for DMA Count 16
          DW        OFFSET DMA_17      ; Entry for DMA Count 17
          DW        OFFSET DMA_18      ; Entry for DMA Count 18
          DW        OFFSET DMA_19      ; Entry for DMA Count 19
          DW        OFFSET DMA_1a      ; Entry for DMA Count 1a
          DW        OFFSET DMA_1b      ; Entry for DMA Count 1b
          DW        OFFSET DMA_1c      ; Entry for DMA Count 1c
          DW        OFFSET DMA_1d      ; Entry for DMA Count 1d
          DW        OFFSET DMA_1e      ; Entry for DMA Count 1e
          DW        OFFSET DMA_1f      ; Entry for DMA Count 1f
          DW        OFFSET exiting     ; Address Of Exit Code
          DW        OFFSET exiting     ; Address Of Exit Code
          DW        OFFSET exiting     ; Address Of Exit Code
          DW        OFFSET exiting     ; Address Of Exit Code
          DW        OFFSET exiting     ; Address Of Exit Code
          DW        OFFSET exiting     ; Address Of Exit Code
          DW        OFFSET exiting     ; Address Of Exit Code
          DW        OFFSET exiting     ; Address Of Exit Code
          DW        OFFSET exiting     ; Address Of Exit Code
          DW        OFFSET exiting     ; Address Of Exit Code
          DW        OFFSET exiting     ; Address Of Exit Code
          DW        OFFSET exiting     ; Address Of Exit Code
          DW        OFFSET exiting     ; Address Of Exit Code
          DW        OFFSET exiting     ; Address Of Exit Code
          DW        OFFSET exiting     ; Address Of Exit Code
          DW        OFFSET exiting     ; Address Of Exit Code
```

As indicated above, the entire software decoding network table is initially set to the address of the "exiting routine." Then depending upon how slow or fast the system clock interrupts a certain number of the lower-indexed entries of the table are set to the address of a processing routine. These processing routines are identical and are sequentially located in the routine so that the software decoding network vector table simply vectors the timer interrupt routine to the first of n sequentially executed processing routines where n represents the number of bytes remaining in the sector transfer. In this way the last few bytes of the sector transfer can be accurately monitored (DMA Shadowing) without significantly affecting overall system performance.

Each of the processing routines, except the last one, performs exactly the same function as indicated below:

```
DMA_1f:
               mov          dx,bx
               mov          cx,5
DMA_1f_A:
```

-continued

```
            in        al,05H
            mov       bl,al
            in        al,05H
            mov       bh,al
            cmp       bx,dx
            jne       DMA_1e
            in        al,05H
            mov       bl,al
            in        al,05H
            mov       bh,al
            cmp       bx,dx
            jne       DMA_1e
            loop      DMA_1f_A
DMA_1e:
            mov       dx,bx
            mov       cx,5
DMA_1e_A:
            in        al,05H
            mov       bl,al
            in        al,05H
            mov       bh,al
            cmp       bx,dx
            jne       DMA_1d
            in        al,05H
            mov       bl,al
            in        al,05H
            mov       bh,al
            cmp       bx,dx
            jne       DMA_1d
            loop      DMA_1e_A
```

-continued

. . .

The above routines represent the code required to completely monitor (shadow) the DMA process by watching the DREQ and DACK signal through the DMA controller. In other words, by watching the DMA controller's register that indicates when a DMA request is active (DREQ) then it is possible to completely monitor the sector transfer. It is not necessary to concern ourselves with the timing between the DREQ and DACK signals until the very last data byte of the transfer. Therefore, the routines above simply "shadow" the DMA process until the last byte at which time it is necessary to invoke the error detection and recovery procedure. When the DMA controller transfer register contains a value of 16 or less then the CPU begins to execute inline code that watches each data byte transfer in terms of signal requests and acknowledgements.

Once the last byte to be transferred has been identified then the DMA_O routine begins the process of determining the actual time taken between the DMA Request (DREQ) and the FDC's DMA Acknowledgement (DACK). This process is presented below:

```
DMA_0
            mov     cx,_TEXT : ratio          ; Up to 32 usec Before DREQ
            shl     cx,1
            shl     cx,1
DMA_DRQ_LO:
            in      al,08H                    ; Read DMA Status Register
            test    al,40H                    ; Is DRQ Active?
            jne     DMA_DRQ_HI                ; If Non-Zero, Then DRQ is Active
            loop    DMA_DRQ_LO                ; Otherwise, Continue Checking
            jmp     exiting                   ; DRQ Not Active so Exit
DMA_DRQ_HI:
            mov     cx,OFFFFH                 ; Load Maximum Loop Count
DMA_DRQ_ACTIVE:
            in      al,08H                    ; Read DMA Status Register
            test    al,04H                    ; Is DRQ Active?
            je      DMA_DRQ_INACTIVE          ; If Zero, Then DRQ is Inactive
            loop    DMA_DRQ_ACTIVE            ; Otherwise, Continues Active
            jmp     exiting                   ; DRQ Stuck High (Terrible Error)
DMA_DRQ_INACTIVE:
            cmp     cx,OFFFFH                 ; Check to See if DREQ Remained Active
            jne     DMA_TIME_ACTIVE           ; If so, Then Compute Time Active
            jmp     exiting                   ; Otherwise, Exit Normally
DMA_TIME_ACTIVE:
            mov     ax,_TEXT : avg            ; Load AX With Average 1 Loop Time
            not     cx                        ; Convert Count Down Value To
                                              ; Times Through The Active Loop
                                              ; (Value Used for "n" in Equation)
            inc     cx                        ; Account for First Time Through Loop
            mul     cl                        ; Compute (n * AVG)
            mov     _TEXT : time_active,ax    ; Computed Time Active (ticks)
            cmp     ax,14                     ; Compare with 12 usec Specification
            ja      DMA_error                 ; If TA > 12usec, Then Issue Error
            jmp     exiting                   ; Otherwise, Continue Normal Operation
DMA_error:
            mov     _TEXT : errors, OFFH      ; Set Error Flag to Indicate the Error
            jmp     exiting                   ; DRQ Not Active so Exit
exiting:
;           pop     ax                        ; Obtain Old 8259 IMR From Stack
;           out     021H,al                   ; Set up Old 8259 IMR
            cmp     _TEXT : CHN1_flag,0       ; Is Channel 1 Disabled?
            jz      not_CHN1                  ; If 0, Then not Disabled
                                              ; Otherwise, Must Be Enable
            mov     al,01                     ; DMA Channel 1 Enable Pattern
            out     OAH,al                    ; Enable DMA Channel 1
not_CHN1:
            mov     si,_TEXT : SI_reg         ; Restore SI Register
            mov     dx,_TEXT : DX_reg         ; Restore DX Register
            mov     cx,_TEXT : CX_reg         ; Restore CX Register
            mov     bx,_TEXT : BX_reg         ; Restore BX Register
            mov     ax,_TEXT : AX_reg         ; Restore AX Register
```

-continued

```
chk_old:
          cmp     _TEXT : ticks,128     ; Check to See if Old INT 8 is Needed
          jb      send_EOI              ; If not Equal, Then Exit Timer ISR
          mov     _TEXT : ticks,0       ; Otherwise, Zero Total Tick Count
          jmp     _TEXT : int8_ptr      ; And Calling Old INT 8 Routine
send_EOI:
          push    ax                    ; Save AX Register
          mov     al,020H               ; 8259 End of Interrupt (EOI)
          out     020H,al               ; Send EOI to 8259
          pop     ax                    ; Restore Ax Register
timer_exit:
          iret                          ; Return from Interrupt
_Timer_isr endp
```

The actual process is similar to the previous routines, however there is a counter that is initialized which represents the time required to perform the code presented above. If this counter expires (counts down to zero) and the transition has not occurred (DREQ to DACK) then the DMA operation is aborted and an error condition is returned which will cause the operation to be retried by the operating system. Furthermore, since the DMA operation was terminated then the adjacent sector is not in danger of being corrupted by the operation.

Thus, through software DMA shadowing, it is possible to determine when the last byte of the transfer is about to be transferred. Therefore, it is possible to disable any hardware and/or software resource that is present in the system that can cause the last data byte's transfer to be delayed. The use of software DMA shadowing accordingly allows system software and device drivers to be used to create a critical region about the last data byte transfer that will ensure that the byte is transferred correctly (and not delayed).

The invention described herein provides a complete software implementation of a device driver that is capable of detecting an undetectable data corruption problem without hardware redesign and/or internal modification to an existing FDC. Furthermore, the unique and innovative approach taken which consists of DMA shadowing and use of a software decoding network allows the implementation of the invention to be small (approximately 3.5 kilobytes in a PC/MS-DOS environment) and only marginally degrade performance (<10%) and only during floppy write operations.

The number of FDCs installed in computer systems today is well over 20 million. In order to solve this problem the vendors of such devices have very few alternatives, of which most are extremely costly. Therefore, a software-only solution to this problem is a significant advance in the computer industry. Moreover, the robustness of the design allows the system and method of the present invention to dynamically adjust to processor speeds that encompass the original IBM Personal Computers executing at 4.77 Mhz to the latest workstations that execute at well over 50 MHz.

The present invention may be embodied in other specific forms without departing from its spirit or essential characteristics. The described embodiments are to be considered in all respects only as illustrative, and not restrictive. The scope of the invention is, therefore, indicated by the appended claims, rather than by the foregoing description. All changes which come within the meaning and range of equivalency of the claims are to be embraced within their scope.

What is claimed and desired to be secured by United States Letters Patent is:

1. A method for detecting and preventing floppy diskette controller data transfer errors in computer systems having:
 a central processing unit (CPU);
 a system interrupt timer;
 a floppy diskette, the floppy diskette having at least one sector for receiving multiple data bytes;
 a floppy diskette controller (FDC) for controlling the transfer of data to the floppy diskette;
 means associated with the FDC for providing a data request (DREQ) signal and a data acknowledge (DACK) signal, the DREQ signal being provided when data transfer is requested and the DACK signal being provided when data transfer is permitted; and
 means for counting data bytes transferred to the floppy diskette, said counting means providing a data transfer byte count,
the method comprising the steps of:
 determining if a requested computer system operation is a floppy diskette write operation;
 reading the data transfer byte count provided by said counting means;
 monitoring data byte transfers to the floppy diskette so as to determine when a last data byte is being transferred to a sector of the floppy diskette;
 measuring time between the data request (DREQ) and data acknowledge (DACK) signals for said last data byte transfer to a sector of the floppy diskette; and
 forcing an error condition if the measured time between said DREQ and DACK signals exceeds a specified value.

2. A method for detecting and preventing floppy diskette controller data transfer errors in computer systems having:
 a central processing unit (CPU);
 a system interrupt timer;
 a floppy diskette, the floppy diskette having at least one sector for receiving multiple data bytes;
 a floppy diskette controller (FDC) for controlling the transfer of data to the floppy diskette;
 means associated with the FDC for providing a data request (DREQ) signal and a data acknowledge (DACK) signal, the DREQ signal being provided when data transfer is requested and the DACK signal being provided when data transfer is permitted; and
 means for counting data bytes transferred to the floppy diskette, said counting means providing a data transfer byte count,
the method comprising the steps of:
 determining if a requested computer system operation is a floppy diskette write operation;

hooking an interrupt vector and directing it to an interposer routine;

reprogramming the system interrupt timer to interrupt faster than normal;

installing and calling a software decoding network call vector;

reading the data transfer byte count provided by said counting means;

monitoring data byte transfers to the floppy diskette so as to determine when a last data byte is being transferred to a sector of the floppy diskette;

measuring time between the data request (DREQ) and data acknowledge (DACK) signals for said last data byte transfer to a sector of the floppy diskette;

forcing an error condition if the measured time between said DREQ and DACK signals exceeds a specified value;

reprogramming the system interrupt timer to interrupt normally; and

unhooking said interrupt vector.

3. A method for detecting and preventing floppy diskette controller data transfer errors as defined in claim **1** wherein said means for counting data bytes comprises a data transfer count register of a direct memory access (DMA) controller and wherein the reading step comprises reading the DMA controller's data transfer count register.

4. A method for detecting and preventing floppy diskette controller data transfer errors as defined in claim **1** further comprising the step of hooking an interrupt vector and directing it to an interposer routine.

5. A method for detecting and preventing floppy diskette controller data transfer errors as defined in claim **1** further comprising the step of reprogramming the system interrupt timer to interrupt faster than normal.

6. A method for detecting and preventing floppy diskette controller data transfer errors as defined in claim further comprising the step of installing and calling a software decoding network call vector.

7. A method for detecting and preventing floppy diskette controller data transfer errors as defined in claim **2** wherein said means for counting data bytes comprises a data transfer count register of a direct memory access (DMA) controller and wherein the reading step comprises reading the DMA controller's data transfer count register.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. :	5,379,414
DATED :	January 3, 1995
INVENTOR(S) :	Phillip M. Adams

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

In column 10, line 34, after "Remarks:", please delete "Timer enable", and insert therefor -- Timer_enable --.

In column 11, line 18, please delete "cloak", and insert therefor -- clock --.

In column 12, line 20, please delete "Sector" and insert therefor -- sector --.

In column 22, line 15, after "claim", please insert -- 1 --.

Signed and Sealed this

Twenty-third Day of May, 1995

Attest:

Bruce Lehman

BRUCE LEHMAN

Attesting Officer

Commissioner of Patents and Trademarks

Exhibit B

US005983002A

United States Patent [19]

Adams

[11] **Patent Number: 5,983,002**

[45] **Date of Patent: Nov. 9, 1999**

[54] **DEFECTIVE FLOPPY DISKETTE CONTROLLER DETECTION APPARATUS AND METHOD**

[75] Inventor: **Phillip M. Adams**, Salt Lake City, Utah

[73] Assignee: **Phillip M. Adams & Associates, L.L.C.**, Salt Lake City, Utah

[21] Appl. No.: **08/729,172**

[22] Filed: **Oct. 11, 1996**

[51] **Int. Cl.**^{6} **G06F 11/263**

[52] **U.S. Cl.** **395/183.18**; 395/183.17; 395/842; 371/62

[58] **Field of Search** 395/183.18, 185.07, 395/183.21, 842, 843, 183.17; 371/21.2, 21.3, 61, 62

[56] **References Cited**

U.S. PATENT DOCUMENTS

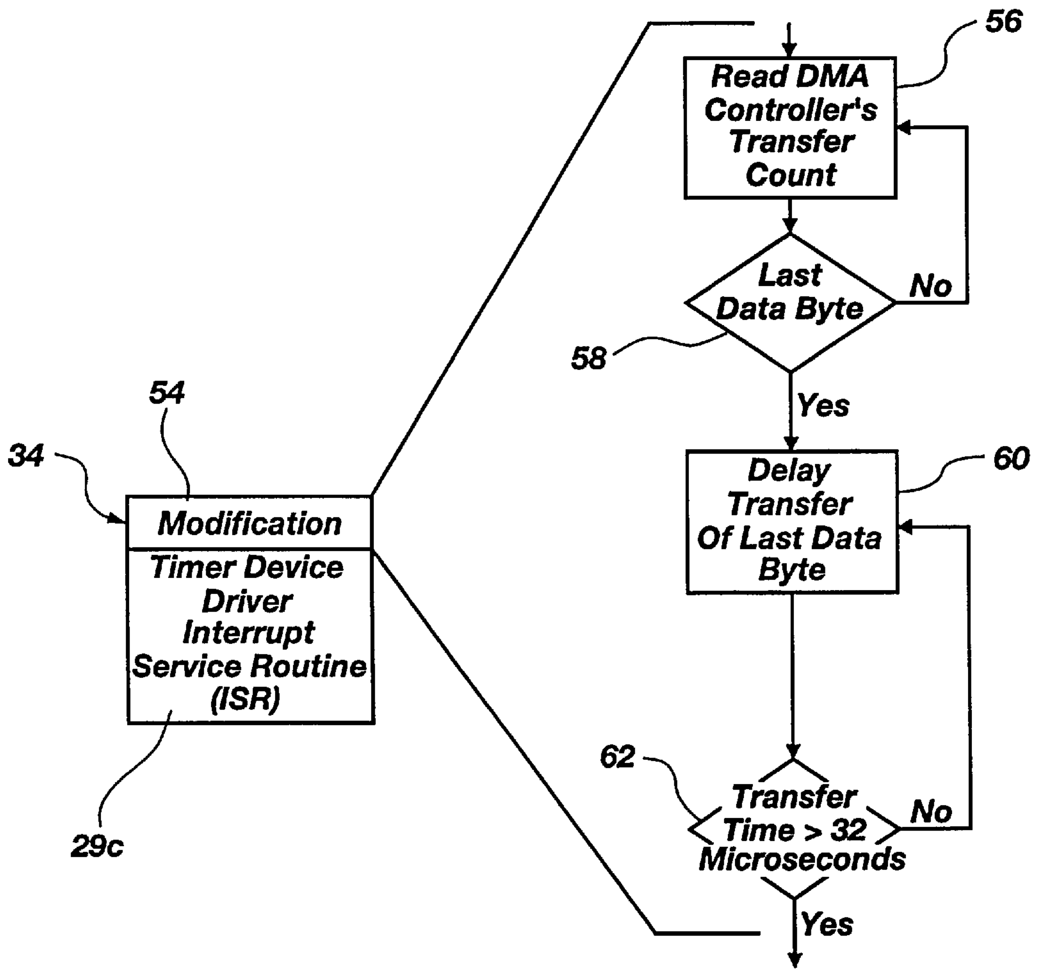
4,789,985	12/1988	Akahoshi et al.	371/11
5,379,414	1/1995	Adams	395/185.08
5,416,782	5/1995	Wells et al.	371/21.2
5,422,892	6/1995	Hii et al.	371/21.2
5,442,753	8/1995	Waldrop et al.	395/842
5,619,642	4/1997	Nielson et al.	395/183.18
5,649,212	7/1997	Kawamura et al.	395/570
5,666,540	9/1997	Hagiwara et al.	395/750.05

Primary Examiner—Robert W. Beausoliel, Jr.
Assistant Examiner—Scott T. Baderman
Attorney, Agent, or Firm—Madson & Metcalf

[57] **ABSTRACT**

A system and method which provides a complete software implementation of a detection process that is capable of detecting defective Floppy Diskette Controllers ("FDCs") without visual hardware inspection or identification. The approach taken includes a multi-phase strategy incorporating programmatic FDC identification, software DMA shadowing, defect inducement, and use of a software decoding network which allows the implementation of the invention to adjust to a wide range of computer system performance levels.

15 Claims, 7 Drawing Sheets


56
Read DMA Controller's Transfer Count
Last Data Byte
No
58
Yes
Delay Transfer Of Last Data Byte
60
54
34
Modification
Timer Device Driver Interrupt Service Routine (ISR)
29c
62
Transfer Time > 32 Microseconds
No
Yes

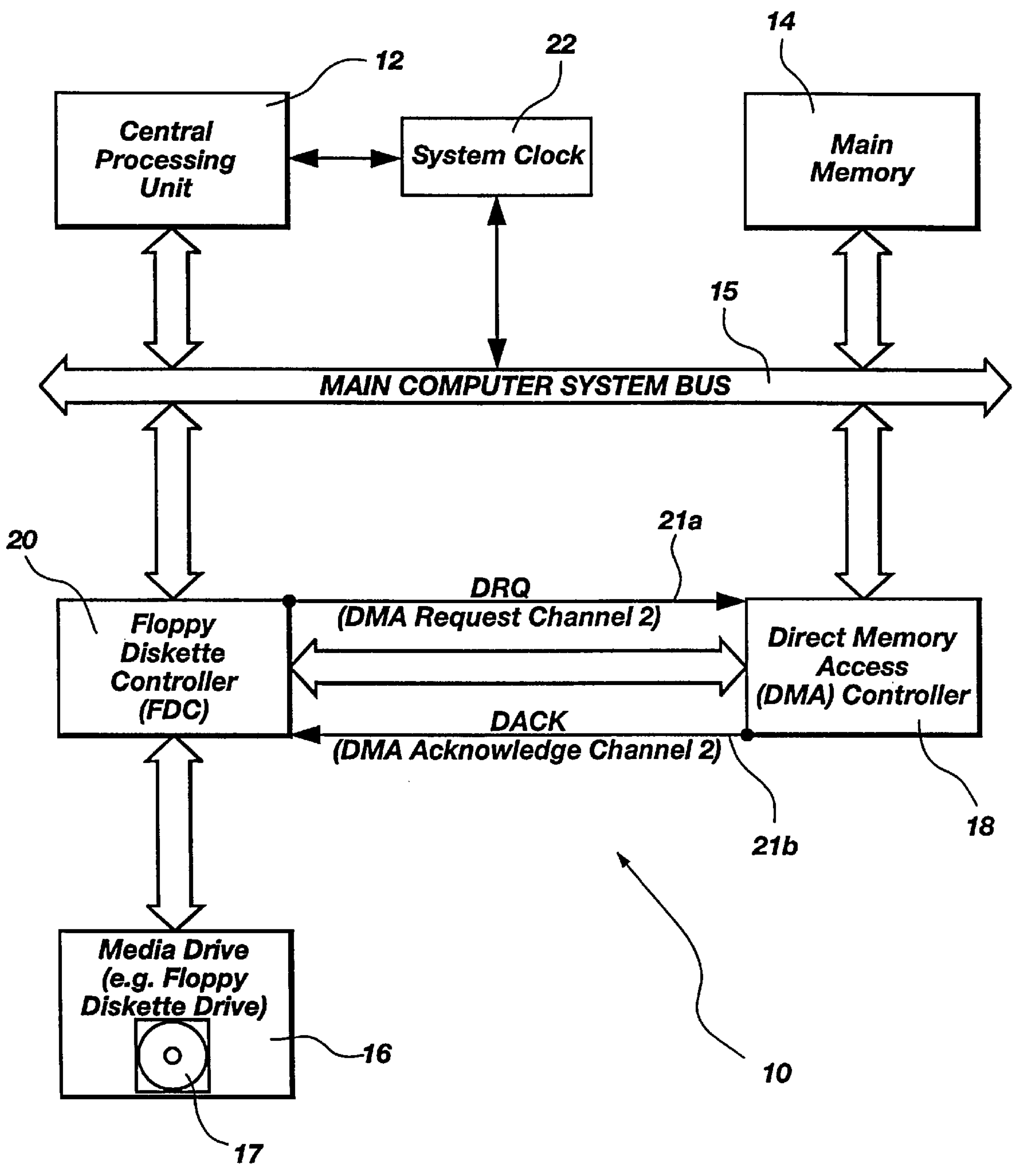

Central Processing Unit
12
System Clock
22
Main Memory
14
15
MAIN COMPUTER SYSTEM BUS
20
Floppy Diskette Controller (FDC)
21a
DRQ (DMA Request Channel 2)
Direct Memory Access (DMA) Controller
DACK (DMA Acknowledge Channel 2)
21b
18
Media Drive (e.g. Floppy Diskette Drive)
16
17
10

Fig. 1

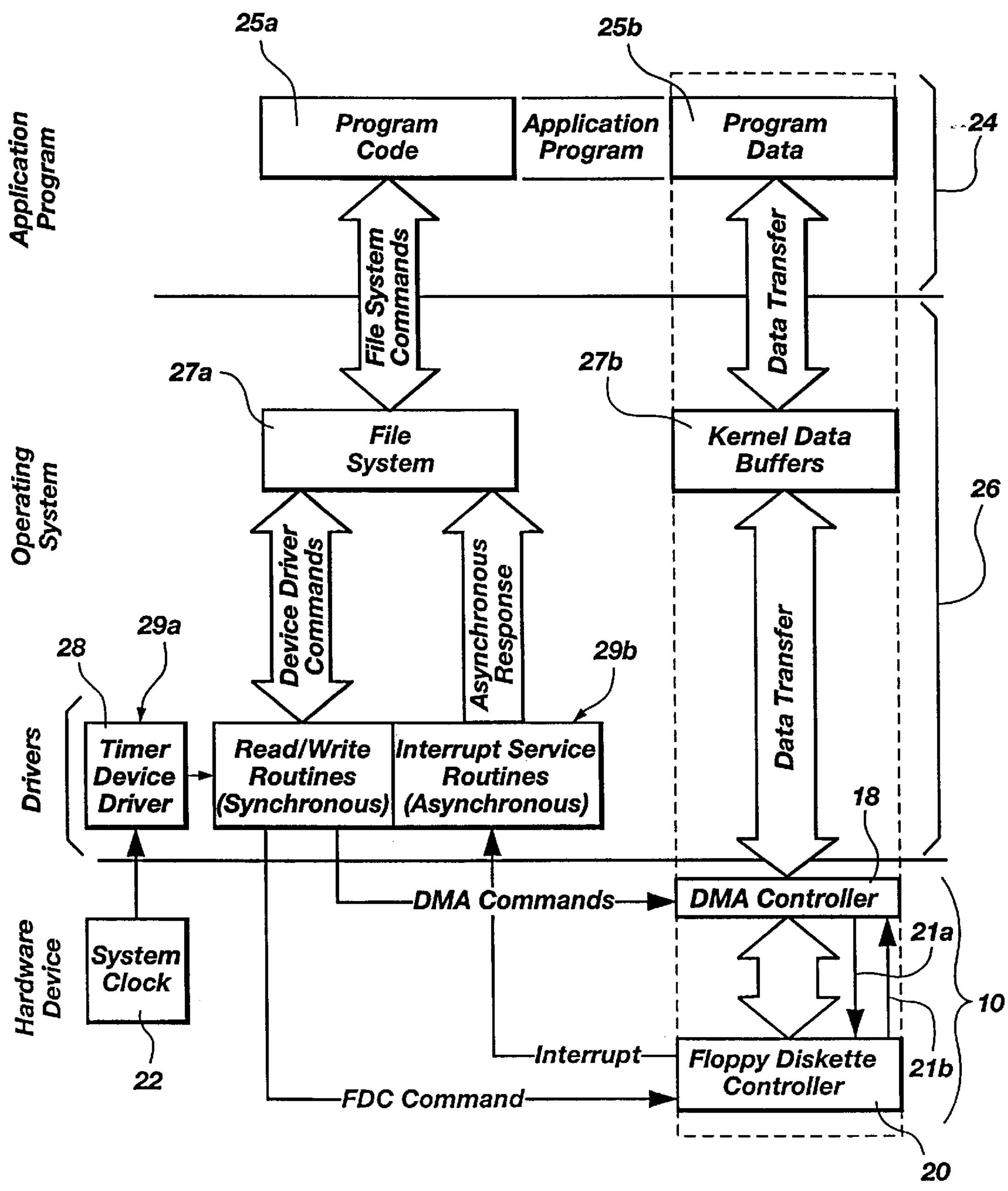

25a
25b
Application Program
Program Code
Application Program
Program Data
24
File System Commands
Data Transfer
27a
27b
Operating System
File System
Kernel Data Buffers
26
Device Driver Commands
Asynchronous Response
Data Transfer
28
29a
29b
Drivers
Timer Device Driver
Read/Write Routines (Synchronous)
Interrupt Service Routines (Asynchronous)
18
DMA Commands
DMA Controller
Hardware Device
System Clock
21a
10
21b
Interrupt
Floppy Diskette Controller
FDC Command
22
20

Fig. 2

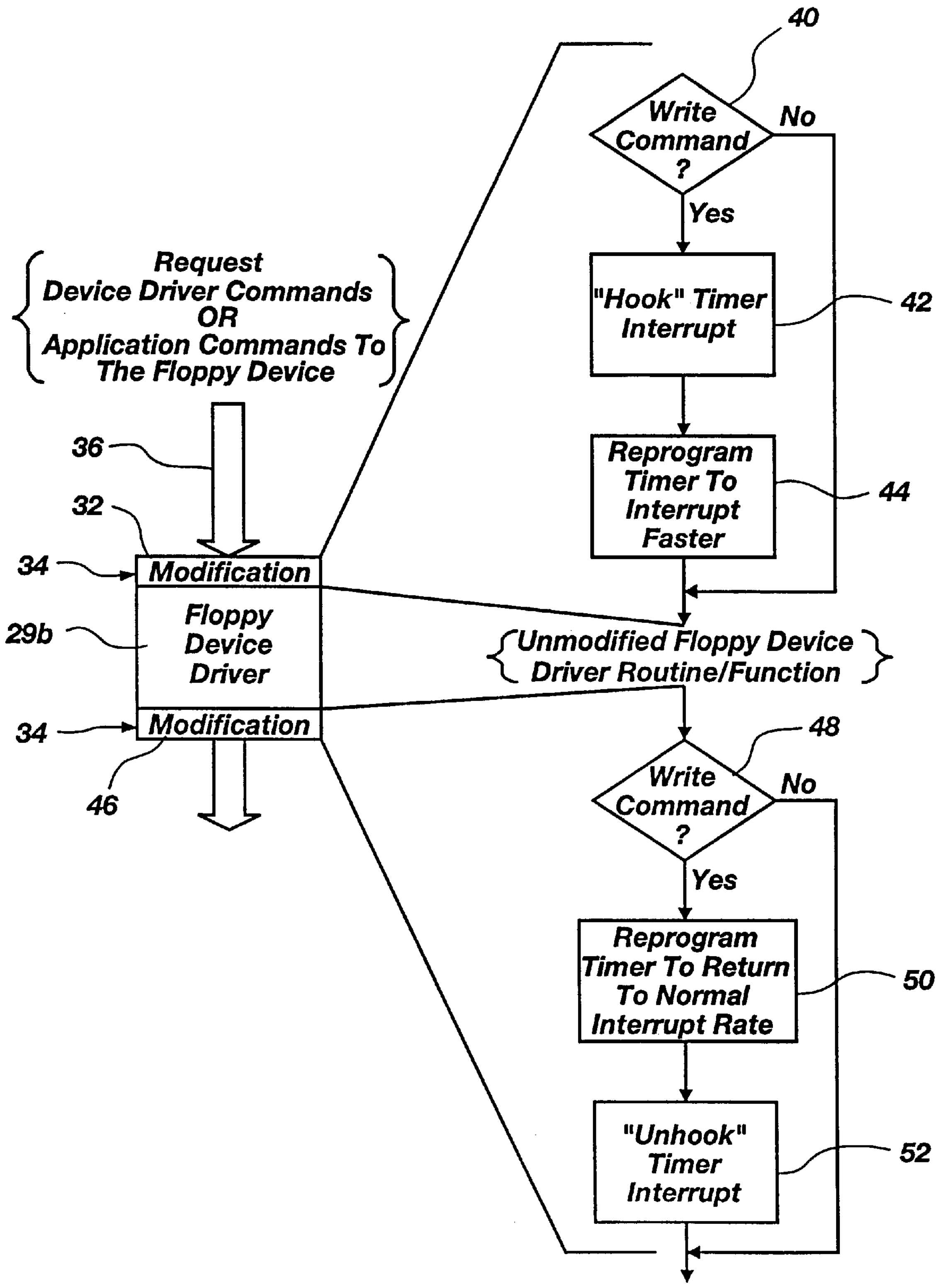

40
Write Command ?
No
Yes
Request Device Driver Commands OR Application Commands To The Floppy Device
"Hook" Timer Interrupt
42
Reprogram Timer To Interrupt Faster
44
36
32
34
Modification
Floppy Device Driver
29b
34
Modification
46
Unmodified Floppy Device Driver Routine/Function
48
Write Command ?
No
Yes
Reprogram Timer To Return To Normal Interrupt Rate
50
"Unhook" Timer Interrupt
52

Fig. 3

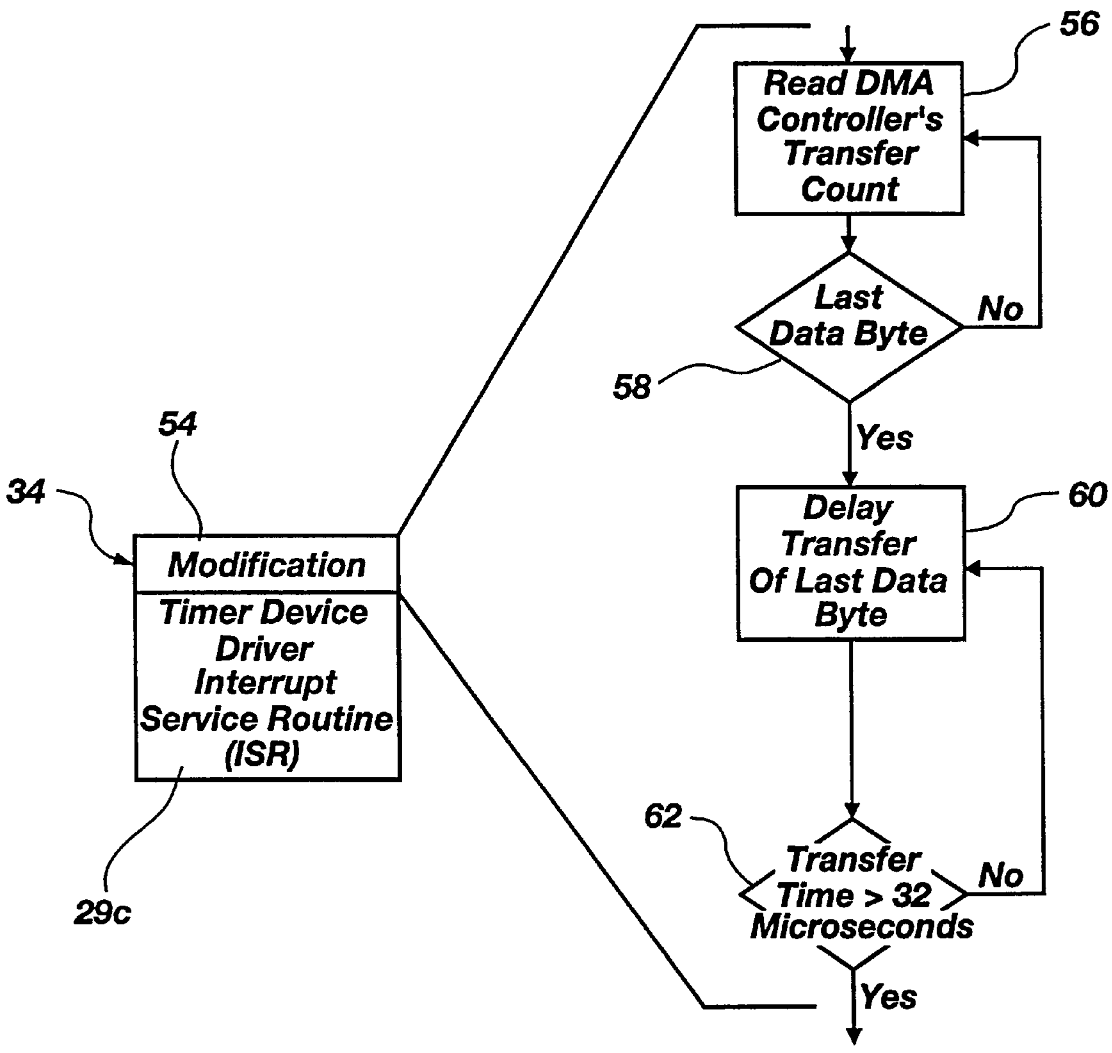

56
Read DMA Controller's Transfer Count
Last Data Byte
No
58
Yes
60
Delay Transfer Of Last Data Byte
54
34
Modification
Timer Device Driver Interrupt Service Routine (ISR)
29c
62
Transfer Time > 32 Microseconds
No
Yes

Fig. 4

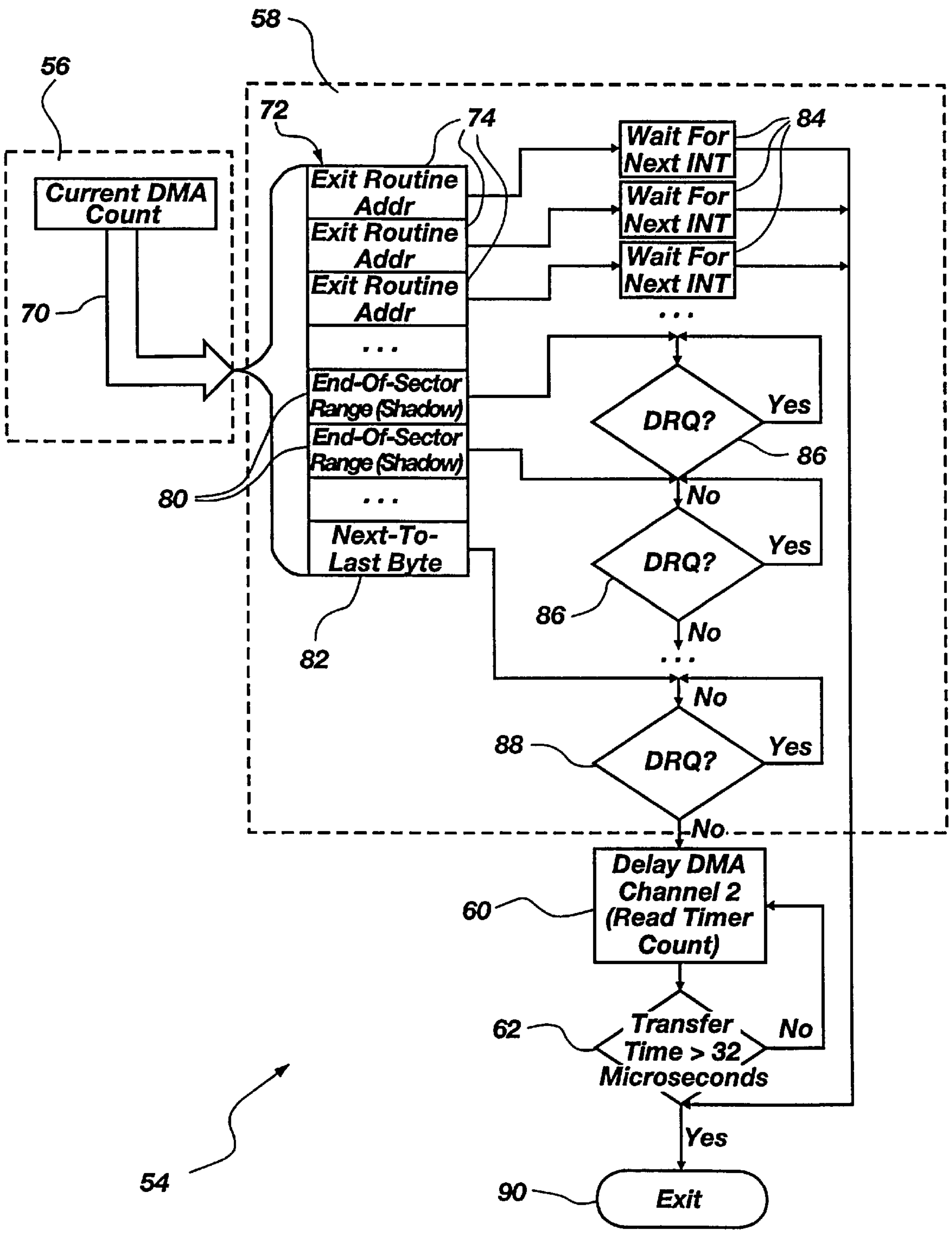
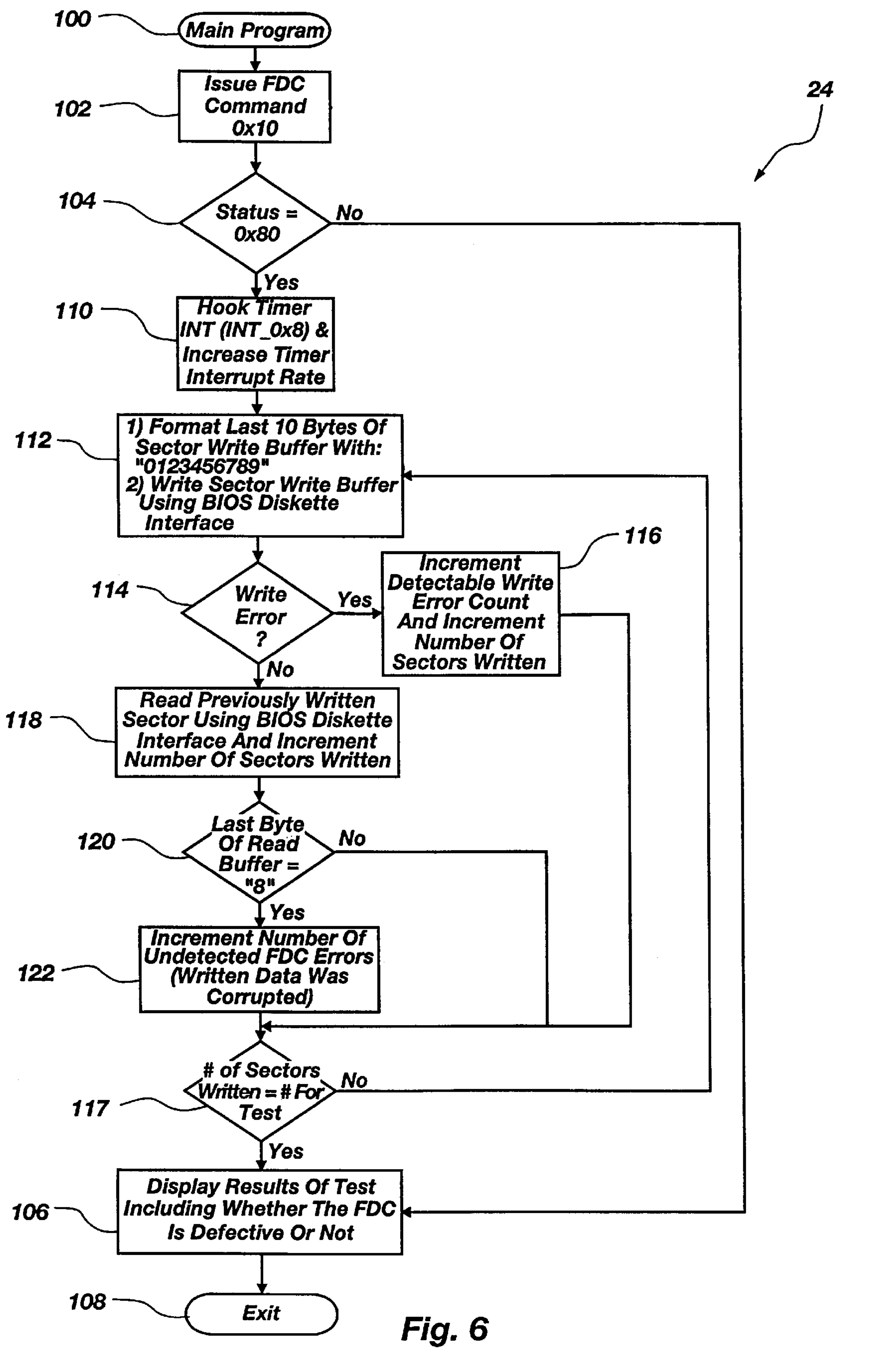

58
56
72
74
84
Current DMA Count
Exit Routine Addr
Exit Routine Addr
Exit Routine Addr
. . .
End-Of-Sector Range (Shadow)
End-Of-Sector Range (Shadow)
. . .
Next-To-Last Byte
Wait For Next INT
Wait For Next INT
Wait For Next INT
. . .
70
80
82
DRQ?
Yes
No
86
DRQ?
Yes
No
86
. . .
No
88
DRQ?
Yes
No
Delay DMA Channel 2 (Read Timer Count)
60
62
Transfer Time > 32 Microseconds
No
Yes
90
Exit
54

Fig. 5



24
100
Main Program
102
Issue FDC Command 0x10
104
Status = 0x80
No
Yes
110
Hook Timer INT (INT_0x8) & Increase Timer Interrupt Rate
112
1) Format Last 10 Bytes Of Sector Write Buffer With: "0123456789"
2) Write Sector Write Buffer Using BIOS Diskette Interface
116
Increment Detectable Write Error Count And Increment Number Of Sectors Written
114
Write Error ?
Yes
No
118
Read Previously Written Sector Using BIOS Diskette Interface And Increment Number Of Sectors Written
120
Last Byte Of Read Buffer = "8"
No
Yes
122
Increment Number Of Undetected FDC Errors (Written Data Was Corrupted)
# of Sectors Written = # For Test
No
117
Yes
106
Display Results Of Test Including Whether The FDC Is Defective Or Not
108
Exit

Fig. 6

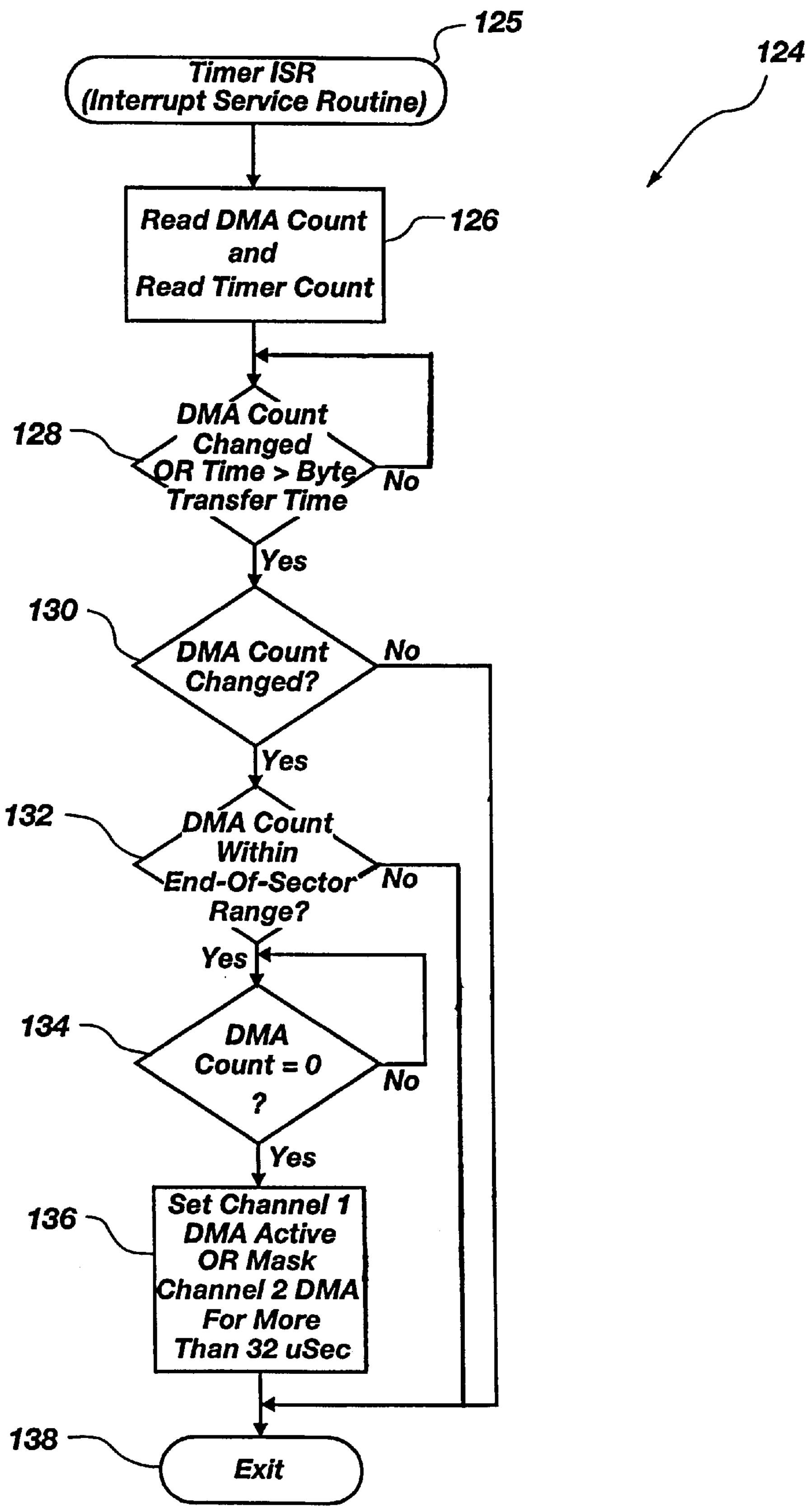

125
124
Timer ISR (Interrupt Service Routine)
Read DMA Count and Read Timer Count
126
128
DMA Count Changed OR Time > Byte Transfer Time
No
Yes
130
DMA Count Changed?
No
Yes
132
DMA Count Within End-Of-Sector Range?
No
Yes
134
DMA Count = 0 ?
No
Yes
136
Set Channel 1 DMA Active OR Mask Channel 2 DMA For More Than 32 uSec
138
Exit

Fig. 7

DEFECTIVE FLOPPY DISKETTE CONTROLLER DETECTION APPARATUS AND METHOD

BACKGROUND

1. The Field of the Invention

This invention relates to the detection of defective Floppy Diskette Controllers ("FDCs") where an undetected data error causes data corruption and, more particularly, to novel systems and methods implemented as a software-only detection mechanism which eliminates the need for visual inspection or identification of the FDCs.

2. The Background Art

Computers are now used to perform functions and maintain data that is critical to many organizations. Businesses use computers to maintain essential financial and other business data. Computers are also used by government to monitor, regulate, and even activate, national defense systems. Maintaining the integrity of the stored data is essential to the proper functioning of these computer systems, and data corruption can have serious (even life threatening) consequences.

Most computer systems include diskette drives for storing and retrieving data on floppy diskettes. For example, an employee of a large financial institution may have a personal computer that is attached to the main system. In order to avoid processing delays on the mainframe, the employee may routinely transfer data files from a host system to a local personal computer and then back again, temporarily storing data on a local floppy diskette. Similarly, an employee with a personal computer at home may occasionally decide to take work home, transporting data away from and back to the office on a floppy diskette.

Data transfer to and from a floppy diskette is controlled by a device called a Floppy Diskette Controller ("FDC"). The FDC is responsible for interfacing the computer's Central Processing Unit ("CPU") with the physical diskette drive. Significantly, since the diskette is spinning, it is necessary for the FDC to provide data to the diskette drive at a specified data rate. Otherwise, the data will be written to a wrong location on the diskette.

The design of an FDC accounts for situations occurring when a data rate is not adequate to support a rotating diskette. Whenever this situation occurs, the FDC aborts the write operation and signals to the CPU that a data underrun condition has occurred.

Unfortunately, however, it has been found that a design flaw in many FDCs makes impossible the detection of certain data underrun conditions. This flaw has, for example, been found in the NEC 765, INTEL 8272 and compatible Floppy Diskette Controllers. Specifically, data loss and/or data corruption may routinely occur during data transfers to or from diskettes (or even tape drives and other media attached via the FDC), whenever the last data byte of a sector being transferred is delayed for more than a few microseconds. Furthermore, if the last byte of a sector write operation is delayed too long then the next (physically adjacent:) sector of the diskette will be destroyed as well.

For example, it has been found that these faulty FDCs cannot detect a data underrun on the last byte of a diskette read or write operation. Consequently, if the FDC is preempted or otherwise suspended during a data transfer to the diskette (thereby delaying the transfer), and an underrun occurs on the last byte of a sector, the following occur: (1) the underrun flag does not get set, (2) the last byte written

to the diskette is made equal to the previous byte written, and (3) a successful Cyclic Redundancy Check ("CRC") is generated on the improperly altered data. The result is that incorrect data is written to the diskette and validated by the FDC.

Conditions under which this problem may occur have been identified in connection with the instant invention by identifying conditions that can delay data transfer to or from the diskette drive. In general, this requires that the computer system be engaged in "multi-tasking" operation or in overlapped input/output ("I/O") operation. Multi-tasking is the ability of a computer operating system to simulate the concurrent execution of multiple tasks.

Importantly, concurrent execution is only "simulated" because only one CPU exists in a typical personal computer. One CPU can only process one task at a time. Therefore, a system interrupt is used to rapidly switch between the multiple tasks, giving the overall appearance of concurrent execution.

MS-DOS and PC-DOS, for example, are single-task operating systems. Therefore, one could argue that the problem described above would not occur. However, a number of standard MS-DOS and PC-DOS operating environments simulate multi-tasking and are susceptible to the problem.

In connection with the instant invention, for example, the following environments have been found to be prime candidates for data loss and/or data corruption due to defective FDCs: local area networks, 327x host connections, high density diskettes, control print screen operations, terminate and stay resident ("TSR") programs. The problem also occurs as a result of virtually any interrupt service routine. Thus, unless MS-DOS and PC-DOS operating systems disable all interrupts during diskette transfers, they are also highly susceptible to data loss and/or corruption.

The UNIX operating system is a multi-tasking operating system. It has been found, in connection with the instant invention, how to create a situation that can cause the problem within UNIX. One example is to begin a large transfer to the diskette and place that transfer task in the background. After the transfer has begun then begin to process the contents of a very large file in a way that requires the use a Direct Memory Access ("DMA") channel of a higher-priority than that of the floppy diskette controller's DMA channel. These might include, for example, video updates, multi-media activity, etc. Video access forces the video buffer memory refresh logic on DMA channel **1**, along with the video memory access, which preempts the FDC operations from occurring on DMA channel **2** (which is lower priority than DMA channel **1**).

This type of example creates an overlapped I/O environment and can force the FDC into an undetectable error condition. More rigorous examples include a concurrent transfer of data to or from a network or tape drive using a high priority DMA channel while the diskette transfer is active. Clearly, the number of possible error producing examples is infinite, yet each is highly probable in this environment.

For all practical purposes the OS/2 and newer Windows operating systems can be regarded as UNIX derivatives. They suffer from the same problems that UNIX does. Two significant differences exist between these operating systems and UNIX.

First, they both semaphore video updates with diskette operations tending to avoid forcing the FDC problem to occur. However, any direct access to the video buffer, i:n either real or protected mode, during a diskette transfer will bypass this feature and result in the same faulty condition as UNIX.

Second, OS/2 incorporates a unique command that tends to avoid the FDC problem by reading back every sector that is written to the floppy diskette in order to verify that the operation completed successfully. This command is an extension to the MODE command (MODE DSKT VER= ON). With these changes, data loss and/or data corruption should occur less frequently than otherwise. However, the FDC problem may still destroy data that is not related to the current sector operation.

A host of other operating systems are susceptible to the FDC problem just as DOS, Windows, Windows 95, Windows NT, OS/2, and UNIX. However, these systems may not have an installed base as large as DOS, Windows, OS/2 or UNIX, and may, therefore, receive less motivation to address the problem. Significantly, as long as the operating systems utilize the FDC and service system interrupts, the problem can manifest itself. This can occur in computer systems that use virtually any operating system.

Some in the computer industry have suggested that data corruption by the FDC is extremely rare and difficult to reproduce. This is similar to the argument presented during the highly publicized 1994 defective INTEL Pentium scenario. Error rate frequencies for the defective Pentium ranged from microseconds to tens-of-thousands of years! The FDC problem is often very difficult to detect during normal operation because of its random characteristics. The only way to visibly detect this problem is to have the FDC corrupt data that is critical to the operation at hand. However, many locations on the diskette may be corrupted, yet not accessed. In connection with the instant invention, the FDC problem has been routinely reproduced and may be more common than heretofore believed.

Computer users may, in fact, experience this problem frequently and not even know about it. After formatting a diskette, for example, the system may inform the user that the diskette is bad, although the user finds that if the operation is performed again on the same diskette everything is fine. Similarly, a copied file may be unusable, and the computer user concludes that he or she just did something wrong. For many in this high-tech world, it is very difficult to believe that the machine is in error and not themselves. It remains typical, however, that full diskette back-ups are seldom restored, that all instructions in programs are seldom, if ever, executed, that diskette files seldom utilize all of the allocated space, and that less complex systems are less likely to exhibit the problem.

Additionally, the first of these faulty FDCs was shipped in the late 1970's. The devices were primarily used at that time in special-purpose operations in which the FDC problem would not normally be manifest. Today, on the other hand, the FDCs are incorporated into general-purpose computer systems that are capable of concurrent operation (multitasking or overlapped I/O). Thus, it is within today's environments that the problem is most likely to occur by having another operation delay a data transfer to a diskette. The more complex a computer system, the more likely it is that one activity will delay another, thereby creating an FDC error condition.

In short, the potential for data loss and/or data corruption is present in all computer systems that utilize the defective version of this type of FDC, presently estimated at about 50 million personal computers. The design flaw in the FDC causes data corruption to occur and manifest itself in the same manner as a destructive computer virus. Furthermore, because of its nature, this problem has the potential of rendering even secure databases absolutely useless.

Various conventional ways of addressing the FDC problem, such as a hardware recall, have significant associated costs, risks and/or disadvantages. In addition to a solution to the FDC problem, an apparatus and method are needed to accurately, rapidly, reliably, and correctly, identify any defective FDC. The identification of defective FDCs is the first step in attempting to solve the problem of defective FDCs. A solution method and apparatus for repairing a defective FDC are disclosed in U.S. Pat. No. 5,379,414 incorporated herein by reference.

BRIEF SUMMARY AND OBJECTS OF THE INVENTION

In view of the foregoing, it is a primary object of the present invention to provide a method and apparatus for detecting defective Floppy Diskette Controllers ("FDCs").

It is another object of the present invention to provide a software (programmatic) solution that may be implemented in a general purpose digital computer, which eliminates the need for visual inspection and identification of the defective FDCs as well as the need for any hardware recall and replacement.

Consistent with the foregoing objects, and in accordance with the invention as embodied and broadly described herein, an apparatus and method are disclosed in one embodiment of the present invention as including data structures, executable modules, and hardware, implementing a detection method capable of immediately, repeatably, correctly, and accurately detecting defective FDCs. The apparatus and method may rely on 1) determining whether or not the FDC under test is a new model FDC (non-defective), and 2) if the FDC under test is not a new model FDC, installing an interposer routine to force the FDC to delay a transfer of a last data byte of a sector either to or from the floppy diskette whose controller is tested. A test condition is thus created in the hardware to cause defective FDCs to corrupt the last data byte of the sector. A second portion of an apparatus and method may confirm a diagnosis. Thus the apparatus and method may ensure that old-model non-defective FDCs are not wrongly identified as defective.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects and features of the present invention will become more fully apparent from the following description and appended claims, taken in conjunction with the accompanying drawings. Understanding that these drawings depict only typical embodiments of the invention and are, therefore, not to be considered limiting of its scope, the invention will be described with additional specificity and detail through use of the accompanying drawings in which:

FIG. **1** is a schematic block diagram of an apparatus illustrating the architecture of a computer system for testing a floppy diskette controller ("FDC")in accordance with the invention;

FIG. **2** is a schematic block diagram illustrating software modules executing on the processor and stored in the memory device of FIG. **1**, including application programs, operating systems, device drivers and computer system hardware such as a floppy diskette;

FIG. **3** is a schematic block diagram of a flow chart depicting one presently preferred embodiment of certain modifications that may be applied to a diskette device driver in order to force an otherwise undetected error condition to occur in a defective FDC, thus enabling the defective FDC detection apparatus and method of the present invention to be activated;

FIG. **4** is a schematic block diagram of a flow chart depicting one presently preferred embodiment of certain modifications that may be made to a timer Interrupt Service Routine ("ISR") to allow timing of a transfer byte's DMA request and DMA acknowledge (DREQ/DACK) cycle in order to ensure that proper conditions exist to create data corruption associated with defective FDCs in accordance with the present invention;

FIG. **5** is a schematic block diagram of a flow chart depicting one presently preferred embodiment of a software decoding network (software vector-table) for use in connection with a defective FDC detection apparatus and method in accordance with the present invention, the software decoding network having one code point/entry for each possible transfer byte in a sector;

FIG. **6** is a schematic block diagram of a flow chart depicting one presently preferred embodiment of an application implementation of the apparatus and method of FIGS. **3** and **4**, wherein a main "driver" portion of an application forces an undetected error condition in a defective FDC enabling activation of a the defective FDC detection system in accordance with the invention; and

FIG. **7** is a schematic block diagram of a flow chart depicting one presently preferred embodiment of certain modifications that may be made to a timer Interrupt Service Routine embedded within the application of FIG. **6** to allow timing of a last byte's DREQ/DACK cycle, ensuring that proper conditions exist to create data corruption associated with defective FDCs in accordance with the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

It will be readily understood that the components of the present invention, as generally described and illustrated in the Figures herein, could be arranged and designed in a wide variety of different configurations. Thus, the following more detailed description of the embodiments of the system apparatus and method of the present invention, as represented in FIGS. **1** through **7**, is not intended to limit the scope of the invention, as claimed, but it is merely representative of the presently preferred embodiments of the invention.

The presently preferred embodiments of the invention will be best understood by reference to the drawings, wherein like parts are designated by like numerals throughout.

The architecture of an apparatus **10**, including computer system implementing one embodiment of the invention is illustrated in FIG. **1**. A Central Processing Unit ("CPU") **12** and main memory **14** may be connected by a bus **15** inside a computer system unit. Instructions (executables) and data structures used by the CPU **12** are kept in main memory **14** during computer work sessions. Main memory **14** is, however, not a permanent storage place for information; it is active only when the apparatus **10** (computer system) is powered up (on). Thus, to avoid losing data, data must be saved on some type of non-volatile storage device. For example, the apparatus may use a "hard disk" storage device permanently installed in the computer system. A computer system **10** may have at least one floppy diskette drive **16** that receives a removable floppy diskette (magnetic storage medium). The floppy diskette likewise may be used for "permanent" (non-volatile) storage of data or software (executables) outside of the computer system **10** flexible (floppy) diskettes are especially useful for transferring data and information between separate computer systems **10**.

In transferring data to a floppy diskette, the CPU **12** may program a Direct Memory Access ("DMA") controller **18** for an input/output ("I/O") transfer. The CPU **12** issues a command to a Floppy Diskette Controller ("FDC") **20** to begin the I/O transfer, and then waits for the FDC **20** to interrupt the CPU **12** with a completion interrupt signal. It is also possible to perform Programmed I/O ("PIO") directly between the CPU **10** and the FDC **20** without involving the DMA controller **18**. This latter approach is seldom used; the majority of computer systems **10** employ DMA for I/O transfers to and from the floppy diskette drive **16**. The invention will thus be described below with particular reference to the DMA controller **18**. If PIO is employed, however, then an I/O transfer is totally controlled by the CPU **12** because the CPU **12** is required to pass each and every data byte to the FDC **16**. As a result, the "DMA shadowing" system and method in accordance with the invention may be directly applied to a PIO data stream. This is readily tractable because the CPU **12** already is controlling the I/O transfer, as will become more readily apparent.

A computer system **10** may have a system clock **22**. The system clock **22** is beneficial when initiating an I/O transfer to the diskette drive **16** because one must not only control the data transfer, but also a drive motor. In this regard, it is important to know when the diskette drive motor has brought a diskette's spin rate up to a nominal RPM required for a data transfer to be successful.

For example, in IBM Personal Computers and "compatibles," the system clock **22** interrupts the CPU **12** at a rate of 18.2 times per second (roughly once every 54.9 milliseconds). This interrupt is used to determine such things as diskette drive motor start and stop time. There are also a host of other time-dependent operations in the computer system **10** that require this granularity of timing.

One presently preferred embodiment of an association between application programs **24** (executables), operating systems **26**, device drivers and hardware is depicted in FIG. **2**. The example presented corresponds to a floppy diskette having a controller **16**.

A system suitable for implementing the invention may include an application program **24** including both executable code **25***a* and associated data **25***b*. The application **24** may interface with the hardware apparatus **10** through an operating system **26**. The operating system may include a file system **27***a* as well as selected buffers **27***b*. The file system **27***a* may include an executable for file system management as well as operating system interfacing. The file system **27***a* may issue commands to drivers **28**.

The drivers **28** may include a timer device driver **29***a*, including a timer ISR, interfacing to the system clock **22**. Likewise, a media drive driver **29***b*, alternatively referred to as a media driver **29***b* may be included. The media driver may interface with a floppy diskette drive or other media drive **16** to maintain persistent storage on media **17**. Although a media drive **16** may typically relate to floppy diskettes, tape drives and other magnetic media may also be used in an apparatus and method in accordance with the invention.

The media driver **29***b* may be responsible for sending instructions and control signals to the media drive controller **20**, which is typically embodied as a floppy diskette controller **20**. Similarly, the media driver **29***b* may instruct and control the DMA controller **18**. The DMA controller manages data transfers between the floppy diskette controller (FDC **20**) and the main memory device **14**. A DMA request (DRQ;DREQ **21***a*) may pass from FDC controller **20** to the direct memory access controller **18** (DMA controller **18**). Likewise, a DMA acknowledge **21***b* or acknowledgement **21***b*, alternatively referred to as a (DACK **21***b*) may be returned from the DMA controller **18** to the FDC **20**.

Referring now to FIGS. **3**–**5**, and more FIGS. **3** and **4**, a method in accordance with the present invention include a module **32**, one of several interposer routines **34**, which is placed between an application's **24** request **36** for floppy service and a floppy device driver **29***b*. The interposer routine **32** is actually a new or modified device driver that forces certain undetected FDC data corruption conditions to exist. As shown, the interposer **32** first tests **40** whether an operation requested **36** is a floppy diskette write operation. Read operations are equally susceptible to the problem and may be used in the detection process, if desired. If so, the major function of the interposer **32** is to insert itself between the application request **36** for floppy service and the floppy device driver **29***b* that will service the request. In a PC/MS-DOS environment, this can be accomplished by "hooking" the INT 0×13 interrupt vector and directing it to the FDD prefix **32** or interposer routine **32**. Reprogramming **44** the timer **22** to interrupt faster (e.g., every 4–7 milliseconds) than normal (e.g. 54.9 milliseconds).

As will become more fully apparent from the following discussion, once a floppy write operation is detected, in a test **40** a software decoding network call vector of the timer interrupt **54** (see FIGS. **4**–**5**) is preferably installed. The current byte count is read **56**, and DMA shadowing **58** begins. When a test **58** shows that a current DMA transfer count (countdown) has reached 0, then the interposer routine **54** delays **60** the DMA transfer of the last byte of the sector transfer. The delay continues until a test **62** determines that the elapsed time is greater than the maximum time required for a data byte to be transferred to the medium **17** (e.g. a low-density diskette; >32 uSec).

This delay **60** forces defective FDCs **20** into an undetected data corruption condition. This condition can be tested **120** by reading back **118** the written data to see whether the last byte or the next-to-the-last byte was actually written to the last byte location of the sector.

Referring again to FIG. **3**, the system clock **22** may be reprogrammed **50** in a suffix routine **46** appended to the floppy device driver **29***b*. The system clock **22** may then interrupt normally (e.g., every 54.9 milliseconds). The timer interrupt **54** is "unhooked" **50** until the test **40** reports the next floppy write operation.

One could allow the timer **22** (clock **22**) to always interrupt at the accelerated rate. Then, a check the timer Interrupt Service Routine ("ISR") **29***c* (see FIG. **4**), within the timer device driver **29***a*, may then determine whether a media (e.g. diskette) write operation is active. Likewise, it is possible to randomly check to see if the last byte of a floppy sector write operation is in progress. However, the foregoing method has superior efficiency and accuracy in creating the condition required for the detection of defective FDCs.

As used herein, "DMA shadowing" may be thought of as programmatic CPU **12** monitoring of data (byte) transfers and timing the last byte of a sector's DREQ **21***a* to DACK **21***b* signals. Importantly, there are, of course, a number of ways of determining when the DREQ **21***a* is present and when the DACK **21***b* is present. The present invention may include the use of any "DMA shadowing" whether the DREQ **21***a* and DACK **21***b* signals are detected at the DMA controller **18**, CPU **12**, system bus **15** or FDC **20**. This includes both explicit means, and implicit means.

For example, inferring the state of the DREQ/DACK cycle is possible from various components in the system that

are triggered or reset from transitions of such signals **21***a*, **21***b*. In one embodiment the DACK **21***b* may cause a Terminal Count ("TC") signal to be asserted by the DMA controller **18**. Therefore, one may imply from the detection of the TC that a DACK **21***b* has occurred.

Whenever an application **24** requests a write operation of the media drive **16**, the system clock **22** may be reprogrammed to interrupt, for example, every 4 to 7 milliseconds. Referring again to FIGS. **4**–**5**, each time the system clock **22** interrupts, the current byte count in the transfer register (countdown register) DMA controller **18** is read **56**. Once the test **58** indicates that the byte counter has reached the last byte, the signal transition from DREQ **21***a* to DACK **21***b* may be timed and accordingly delayed **60**. This transition may be forced to be greater than the maximum time required to transfer one data byte as indicated in the test **62**.

Therefore, defective FDCs **20** are forced into an undetected data corruption state. This state may be detected by writing known data patterns to the next-to-the-last and the last data bytes. Reading the data back will reveal which of the two data bytes was stored in the last byte of the sector. Finally, it is possible to also detect defective FDCs **20** by significantly increasing the delay time during the transfer of the last byte of a sector. This forces the next physically adjacent sector to be zeroed out except for the first byte of that sector.

For the system to maintain proper operation, an interposer routine **34** should save the original INT 0×13 (Hex 13th interrupt vector) contents (address of the original INT 0×13 Interrupt Service Routine) and invoke the original when necessary. Additional aspects of the interposer function **34** are discussed below in connection with other features of the device driver **29***b*.

This implementation of the apparatus and method of the present invention is contemplated for use on an IBM Personal Computer running the PC/MS-DOS operating system. Versions have, however, been developed to operate in the Windows, OS/2 and UNIX environments and may be embodied for other operating systems. The invention is not limited to use with any particular operating system, and adaptations and changes which may be required for use with other operating systems will be readily apparent to those of ordinary skill in the art.

As depicted graphically in FIG. **4** below, a timer ISR routine **29***c* is used for servicing the accelerated interrupt rate of the system clock **22**. The reason that the system clock interrupt rate is accelerated is that during a normal 512 byte data transfer (the typical sector size) 16 microseconds are required for each data byte to be transferred to the FDC (High Density Diskette Mode). Therefore, a typical sector transfer requires 512 times 16 microseconds, or 8,192 microseconds. If the diskette is a low density diskette then the sector transfer time is doubled to 16,384 microseconds (512 times 32 microseconds) because the FDC has half of the amount of data to store in the same rotational time frame (typically 360 RPM).

Referring to FIG. **5**, the timer ISR routine **29***c* within the timer device driver **29***a* with its prefix **54** performs checks on the system **10** to determine if the system **10** is actually transferring data to the FDC **20**. If a sector transfer is not in progress then the timer ISR prefix **54** exits immediately. However, if a sector transfer is in progress then the timer ISR prefix **54** obtains the remaining byte count of the sector transfer **70** and vectors (jumps) through the software decoding network **72** (DMA count table **72**) to an appropriate processing routine **84**, **86**, **88**.

Although the steps **56**, **58** of the module **54** may be implemented with the timer **22** continually interrupting every 8, 16, or 32 microseconds. This level of interrupts may totally consume a PC's processing power, and on most PCs could not be sustained. Thus, in order to perform I)MA shadowing without affecting the total system performance it is important to allow normal operations to continue as usual. It is desirable to have an interrupt (the system clock **22**) that will interrupt close to the end of the sector transfer so that the DREQ **21***a* to DACK **21***b* timing may be determined on the last byte of the sector transfer.

Thus, it is possible to DMA shadow **58** all 512 bytes during a sector transfer, but that would cause the CPU to be totally consumed during the entire sector transfer time. The potential of losing processing activities elsewhere in the system are greatly increased, as in serial communications. Therefore, the clock interrupt routine **29***c* or method **29***c* of FIG. **5** may reduce the CPU involvement to a bare minimum during those floppy write operations with DMA Shadowing. Significantly, the timing may be adjusted to any number of bytes of a sector transfer, from a few bytes to the entire sector count.

One operation performed in the timer ISR routine **29***c* is to vector through the software decoding network **72** to the appropriate processing routine **84**, **86**, **88**. This process is illustrated graphically in FIG. **5**. The software decoding network **72** (software vector-table **72**) has one code point/entry **74**, **80**, **82** for each possible transfer byte in the sector.

The timer interrupt rate can now be in terms of 10's or 100's of byte transfer times. The vector table **72** may cause the program execution of the CPU **12** to enter a cascade **86** of DREQ **21***a*/DACK **21***b* checks only when the transfer (sector) will complete prior to the next timer interrupt. In short, the first entries **74** in the vector table **72** will return **84**, since another timer interrupt will occur before the sector transfer completes. The latter entries **80**, within the desired range, will cascade **86** from one DREQ **21***a*/DACK **21***b* detection to another (shadowing **58** the DMA transfers) until the last byte is transferred.

On the last byte being transferred, the data byte may be delayed by either activating a higher priority DMA **18** channel or masking the DMA channel of the FDC **20**. Although these two techniques are the simplest to program, numerous alternatives may be used to delay **60** data transfers on the DMA **18** channel of the FDC **20**, in accordance with the invention

This software decoding network process **54** is the fastest known software technique for decoding and executing time-dependent situations. Space in the memory space **14** (e.g. the software decoding network vector table **72**) is traded for processing time, the amount of time it would take for one routine to subsume all functionality encoded in each of the routines **84**, **86**, **88** vectored to through the software decoding network vector table **72**.

As indicated above, the entire software decoding network table **72** may be initially set to the address of an "exiting routine **84**." Then depending upon how slow or fast the system clock **22** interrupts, a certain number of the lower-indexed entries **80** of the table **72** may be set to the address of a processing routine **86**. These processing routines **86** may be identical and sequentially located in the routine **54**. Thus, the software decoding network vector table **72** may simply vector the timer ISR routine **29***c* within the driver **29***a* to the first of n sequentially executed processing routines. Here, n represents the number of bytes remaining in the sector transfer. In this way the last few bytes of the sector transfer can be accurately monitored (DMA Shadowing **58**) without significantly affecting overall system performance.

Each of the processing routines **86**, except the last one **88**, may perform exactly the same function. It is not necessary to be concerned with the timing between the DREQ **21***a* and DACK **21***b* signals until the very last data byte of a transfer. Therefore, the routines **86**, **88** above "shadow" **58** the operation of the DMA until the last byte (e.g. corresponding to entry **82** of the vector **72**) at which time the DMA channel of the FDC **20** is delayed as previously described.

Thus, through software DMA shadowing, it is possible to reliably determine when the last byte of the transfer is about to be transferred. Therefore, it is possible to force the last data byte's transfer to be delayed. An alternative approach may include a specialized application program **24** to control all aspects of the operation of the media drive **16**, e.g. floppy diskette drive **16**. This may include a transfer delay of a last byte, as indicated in FIGS. **6** (main application) and **7** (timer interrupt service routine) All aspects of the previous approach may be present. However, here they may be collected into a single application program **24** performing the required functions. The application program **24** may reprogram the system clock to interrupt at an accelerated rate and services the interrupt itself. The application program may then begin a repeated set of diskette write operations using the BIOS interface interrupt (0×13) and then read the written sectors back. Once the sector has been written and read back the data is compared to determine whether or not an undetected error has occurred. A running total of both detected and undetected errors may be output to a display.

Referring now to FIG. **6**, an application **24** may begin at an entry point **100** leading to an initial command **102**. Command **102** is effective to request of a floppy diskette controller (FDC) **20** an identification. A status return of 0×90 (hexadecimal 90) indicates that a FDC **20** is not defective. Alternatively, the command **102** may give rise to a status return of 0×80 hexadecimal 80. This return does not guarantee that an FDC **20** is not defective.

Thus, a test **104** determines whether or not the status of an FDC **20** is hex 80. A negative response may advance the application **24** to a display step **106**. The display **106** may output results of the application **24**. Results may include an indication of whether the FDC **20** being tested is defective or not. Accordingly, a status not equal to a hex 80 results in the test **104** signifying that an FDC **20** is not defective. The step **108** thereafter exits the application **24**.

A positive response to the test **104** advances the application **24** to a hook **110**. The hook **110** is effective to interpose a timer prefix **124** (see FIG. **7**) corresponding the prefix **34** of FIG. **3**, to be installed to operate at the beginning of a timer ISR **29***c* within the timer device driver **29***a*.

A test pattern **112** may format the last few (for example, 10) bytes of a sector write buffer **27***b*. Any known pattern may suffice, for example, a sequential list of all digits from zero to nine may be used. Importantly, the last two digits in such a sequence should be distinct. Thus, a string "0123456789" may provide a test pattern to be written in the last ten bytes of a sector. The test pattern may then be written from a buffer **27***b* to a medium **17** using the BIOS interface for the medium **17** and medium drive **16**.

Following the test pattern **112**, a test **114** may determine whether or not a write error has occurred in writing the buffer **27***b* to the medium **17**. A positive response to the test **114** results in an increment step **116**. The increment **116** tracks the number of successful detections of errors. Thus, the increment **116** indicates that another write error was successfully detected by the FDC **20**. Accordingly, the application **24** may advance from the increment **116** to a test **117**. A test **117** may determine the number of sectors to which the FDC **20** has attempted to write. If the response to the comparison of the test **117** is positive, then all tests are completed and the display step **106** follows. On the contrary, a negative response to this test **117** returns the application **24** to the test pattern **112**, initiating another test cycle.

A negative response to the test **114** indicates that a write error, known to exist, was undetected by the FDC **20**. Accordingly, a negative response to the test **114** advances the application **24** to a read **118**. The read **118** reads back the last previously written sector, using the BIOS diskette interface, such as the driver **29***b*. The step **118** may then increment the number corresponding to sectors that the FDC **20** has attempted to write.

The application **24** may next advance to the test **120** to determine whether the last byte that th e read step **118** has read back from the written sector to a buffer **27***b* is the last, or the next-to-last element of the test pattern from the test pattern step **112**. That is, for example, in the Example above, the test **120** determines whether or not the last byte read back to the buffer **27***b* from this sector being tested is "8" a value of "8" indicates that the FDC has failed to write the tenth element of the test pattern into the last byte location of the sector. This indicates that the FDC has not indicated a write error in the test **114**, and yet has produced the error detected by the test **120**. Thus, the last sector written is corrupted.

A negative response to the test **120** indicates that the last byte was not incorrectly written. Accordingly, the application **24** may advance to the test **117** to determine whether or not the testing is completed. A positive response to the test **120** results in an increment step **122**. The increment step **122** advances the count of undetected errors found during the operation of the FDC **20** during the testing in question. Thus, a step **122** results in a corruption count for sectors attempted to be written by the FDC **20**.

Referring now to FIG. **7**, and also cross-referencing to FIG. **6**, the hook step **110** may install a prefix **54** to a timer ISR **29***c* within the timer device driver **29***a* (see FIG. **4**). The hook **110** interposes the prefix **124** corresponding to the prefix **54** of FIG. **4**, after a call **125** or entry point **125** to the timer ISR **29***c* within the timer device driver **29***a*. Accordingly, whenever the timer ISR **29***c* within the timer device driver **29***a* is called, the prefix **124** will be run before any executables in the timer ISR **29***c* within the timer device driver **29***a*.

The prefix **124** may begin with a read **126** affective to determine a count corresponding to the number of bytes, or a countdown of the remaining bytes, being transferred by the DMA controller **18** from the main memory **14**, through the buffer **27***b* to the FDC **20**. The read **126** may also include a reading of a count (a tick count) of a timer **22** or system clock **22**.

Following the read **126**, a test **128** may determine whether or not an operation is in process affecting the FDC **20**. The FDC **20** is in operation if a count kept by the DMA controller **18** has decremented (changed) within an elapsed time corresponding to the maximum time required for a byte to be transferred. If no change has occurred during that elapsed time, then one may deduce that no activity is occurring. Accordingly, a negative response to the test **128** results in reexecution of the test **128**. Reexecution of the test **128** may continue until a positive response is obtained. Inasmuch as the application **24** is executing a write during the test pattern

112, an eventual positive response to the test **128** is assured. In one embodiment of an apparatus and method in accordance with the invention, the first byte transferred may typically be detected.

A positive response to the test **128** advances the prefix routine **124** to a test **130** to test the countdown or count of the DMA controller **18**. The test **128** corresponds to detection of activity, whereas the test **130** corresponds to iteration of a shadowing process.

The test **130**, whenever a negative response is received, may advance the prefix routine **124** to the exit **138**.

On the other hand, a positive response to the test **130** advances the prefix routine **124** to a test **132** effective to evaluate whether or not the countdown is within some selected range at the end of a sector. A negative response to the test **132** indicates that the countdown is not within some desired end-of-sector range, so the prefix routine **124** should exit **138** without waiting longer. That is, interrupts will continue to occur with a frequency that will detect the desired range at the end of the sector being tested.

A positive response to the test **132** advances the prefix routine **124** to a test **134** for detecting the last byte to be transferred in a sector. If the DMA controller **18** is not counting the last byte to be transferred, then the test **134** may simply continue to test. When the countdown of the DMA controller **18** reaches a value of zero, a positive response to the test **134** advances to a delay step **136**.

The delay step **136** corresponds to the delay **60** illustrated in FIGS. **4**–**5**. The delay **136** may be implemented by preempting a channel over which the DMA controller **18** is communicating with the FDC **20**. For example, a first channel may be made active by some process, thus, overwriting communication over some channel having lesser priority, and corresponding to the FDC **20**. Likewise, the channel corresponding to the DMA communication with the FDC **20** may be masked (suspended) until the time elapsed for the transfer of the data to the sector has exceeded the maximum time permitted for such transfer. Thus, any and all opportunities for writing the last byte to the sector had expired. Thus, an error condition has been assured. Once the delay **136** has assured an error condition the exit **138** returns control of the processor **12** to the non-interrupted processing state.

The invention described heretofore provides detection solution that may be completely implemented in software as a device driver **29***b* that is capable of detecting defective FDCs **20** without visual inspection and identification of the FDCs. Furthermore, the unique and innovative approach taken, relying on DMA shadowing and use of a software decoding network, allows the implementation of the invention to accurately and correctly detect defective FDCs even when non-defective old-model FDCs are involved. Simply stated, it is not sufficient to determine whether the FDC under test is an old or new model FDC. Various vendors manufactured old-model FDCs that are not defective. Therefore, a two-phase detection process may correctly determine whether or not the FDC under test is defective.

The number of FDCs installed in computer systems today is well over 100 million. In order to identify defective FDCs vendors and consumers which have defective FDCs **20** installed have very few alternatives (e.g. recalls; replacement), of which most are extremely costly, for determining whether or not their systems are susceptible to the data corruption presented by defective FDCs **20**. Therefore, an apparatus and method that may be implemented as a software-only solution to this problem is a significant advance in the computer industry. Moreover, the robust design allows the apparatus and method of the present invention to dynamically adjust to processor speeds that encompass the original IBM Personal Computers executing at 4.77 MHz to the latest workstations that execute at well over 200 MHZ.

The present invention may be embodied in other specific forms without departing from its spirit or essential characteristics. The described embodiments are to be considered in all respects only as illustrative, and not restrictive. The scope of the invention is therefore, indicated by the appended claims, rather than by the foregoing description. All changes which come within the meaning and range of equivalency of the claims are to be embraced within their scope.

What is claimed and desired to be secured by United States Letters Patent is:

1. An apparatus for detecting a defective floppy diskette controller, the apparatus comprising:

a processor executing detection executables effective to determine an underrun error undetected by a floppy diskette controller and effective to identify the floppy diskette controller as defective;

a memory device operably connected to the processor to store the detection executables and corresponding detection data;

a system clock operably connected to the processor to provide a time base;

a media drive comprising storage media for storing data;

the floppy diskette controller operably connected to the media drive to control formatting and storage of data on the storage media; and

a direct memory access controller operably connected to the floppy diskette controller and the memory device to control transfers of data between the memory device and the floppy diskette controller.

2. The apparatus of claim **1** wherein the detection executables are effective to cause an underrun error.

3. The apparatus of claim **2** wherein the detection executables cause the underrun error by delaying a transfer of data between the direct memory access controller and the floppy diskette controller.

4. The apparatus of claim **3** wherein the underrun error comprises a delay in transferring a last byte in the transfer.

5. The apparatus of claim **3** wherein the detection data comprises a test pattern.

6. The apparatus of claim **5** wherein the underrun error comprises the test pattern incorrectly copied onto the storage media.

7. The apparatus of claim **1** wherein the detection executables further comprise a prefix routine effective to hook a floppy device driver operating on the processor to control the floppy diskette controller.

8. The apparatus of claim **1** wherein the detection executables are integrated into an application directly loaded and executed on the processor.

9. The apparatus of claim **8** wherein the application is effective to determine on demand whether the floppy diskette controller is susceptible to undetected underrun errors.

10. The apparatus of claim **1** wherein the detection executables include a shadowing executable effective to determine when a last byte is to be transferred from the direct memory access controller to the floppy diskette controller.

11. A memory device operably connected to a processor, a direct memory access controller, a floppy diskette controller controlled by the direct memory access controller, and a media drive controlled by the floppy diskette controller, the memory device storing blocks of data comprising:

a test pattern;

detection executables effective to be run on the processor to force and detect an underrun error not detected by the floppy diskette controller; and

a readback buffer to store a copy of the test pattern read back from the media drive.

12. A method for detecting an underrun error undetected by a floppy diskette controller, the method comprising the steps of:

writing a source test pattern from a memory device to storage media in a media drive controlled by the floppy diskette controller;

interrupting the writing step;

delaying a transfer of a last byte of the source test pattern to the floppy diskette controller to create the underrun error;

completing the writing step;

verifying whether the floppy diskette controller detected the underrun error.

13. The method of claim **12** further comprising reading back to the memory device a written test pattern corresponding to the source test pattern written during the writing step.

14. The method of claim **13** further comprising verifying whether the underrun error occurred in the writing step by checking the last byte of the written test pattern.

15. An apparatus for detecting a defective floppy diskette controller, the apparatus comprising:

a processor executing detection executables effective to precipitate and detect an underrun error undetected by a floppy diskette controller and effective to identify the floppy diskette controller as a defective floppy diskette controller;

a memory device operably connected to the processor to store the detection executables and corresponding detection data;

a system clock operably connected to the processor to provide a time base;

a media drive comprising storage media for storing data;

the floppy diskette controller operably connected to the media drive to control formatting and storage of data on the storage media; and

a direct memory access controller operably connected to the floppy diskette controller and the memory device to control transfers of data between the memory device and the floppy diskette controller.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO.	: 5,983,002	Page 1 of 1
DATED	: November 9, 1999	
INVENTOR(S)	: Philip M. Adams	

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 9,
Line 5, please change "l)MA" to --DMA--.

Column 11,
Line 19, please change "th e" to --the--.
Line 50, please change "affective" to --effective--.

Signed and Sealed this

Twenty-sixth Day of June, 2001

Attest:

Nicholas P. Godici

Attesting Officer

NICHOLAS P. GODICI
Acting Director of the United States Patent and Trademark Office

Exhibit C

US006195767B1

(12) **United States Patent**
Adams

(10) **Patent No.: US 6,195,767 B1**
(45) **Date of Patent: Feb. 27, 2001**

(54) DATA CORRUPTION DETECTION APPARATUS AND METHOD

(76) Inventor: **Phillip M. Adams**, 1466 Chandler Dr., Salt Lake City, UT (US) 84103

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/152,802**

(22) Filed: **Sep. 14, 1998**

(51) **Int. Cl.**^{7} **H02H 3/05**; G06K 5/04
(52) **U.S. Cl.** **714/47**; 714/48; 714/42; 714/43; 714/718
(58) **Field of Search** 714/47, 48, 42, 714/43, 718

(56) **References Cited**

U.S. PATENT DOCUMENTS

3,908,099		9/1975	Borbas et al.	179/175.2
4,942,606		7/1990	Kaiser et al.	380/4
4,996,690		2/1991	George et al.	371/37.1
5,093,910		3/1992	Tulpule et al.	395/575
5,210,860	*	5/1993	Pfeffes et al.	395/575
5,212,795		5/1993	Hendry	395/725
5,233,692		8/1993	Gajjar et al.	395/325
5,237,567		8/1993	Nay et al.	370/85.1
5,379,414	*	1/1995	Adams	395/575
5,416,782		5/1995	Wells et al.	371/21.2
5,422,892		6/1995	Hii et al.	371/24
5,619,642		4/1997	Nielson et al.	395/182.04
5,805,788	*	9/1998	Johnson	395/182.04
5,844,911	*	12/1998	Schadess et al.	371/10.2
5,983,002	*	11/1999	Adams	395/183.18
6,115,199	*	9/2000	Bang	360/51

OTHER PUBLICATIONS

NEC Electronics, Inc., "IBM–NEC Meeting for μPD765A/ μPD72065 Problem" (U.S.A., May 1987). 6 pps.

Intel Corporation, Letter to customers from Jim Sleezer, Product Manager, regarding FDC error and possible solutions (U.S.A., May 2 1988).

Adams, Phillip M., Nova University, Department of Computer Science, "Hardware–Induced Data Virus," Technical Report TR–881101–1 (U.S.A., Nov. 14, 1988).

Advanced Military Computing, "Hardware Virus Threatens Databases," vol. 4, No. 25, pp. 1 & 8 (U.S.A., Dec. 5, 1988).

Intel Corporation, "8237A/8237A–4 /8237A–5 High Performance Programmable DMA Controller" (U.S.A., date unknown.).

Intel Corporation, "8272A Single/Double Density Floppy Disk Controller" (U.S.A., date unknown).

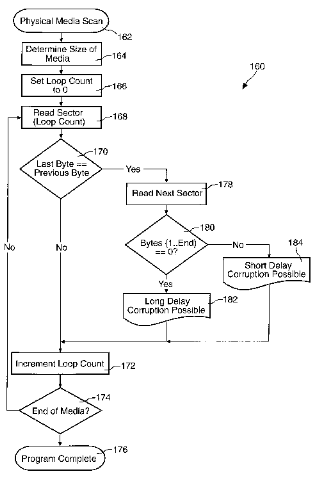
* cited by examiner

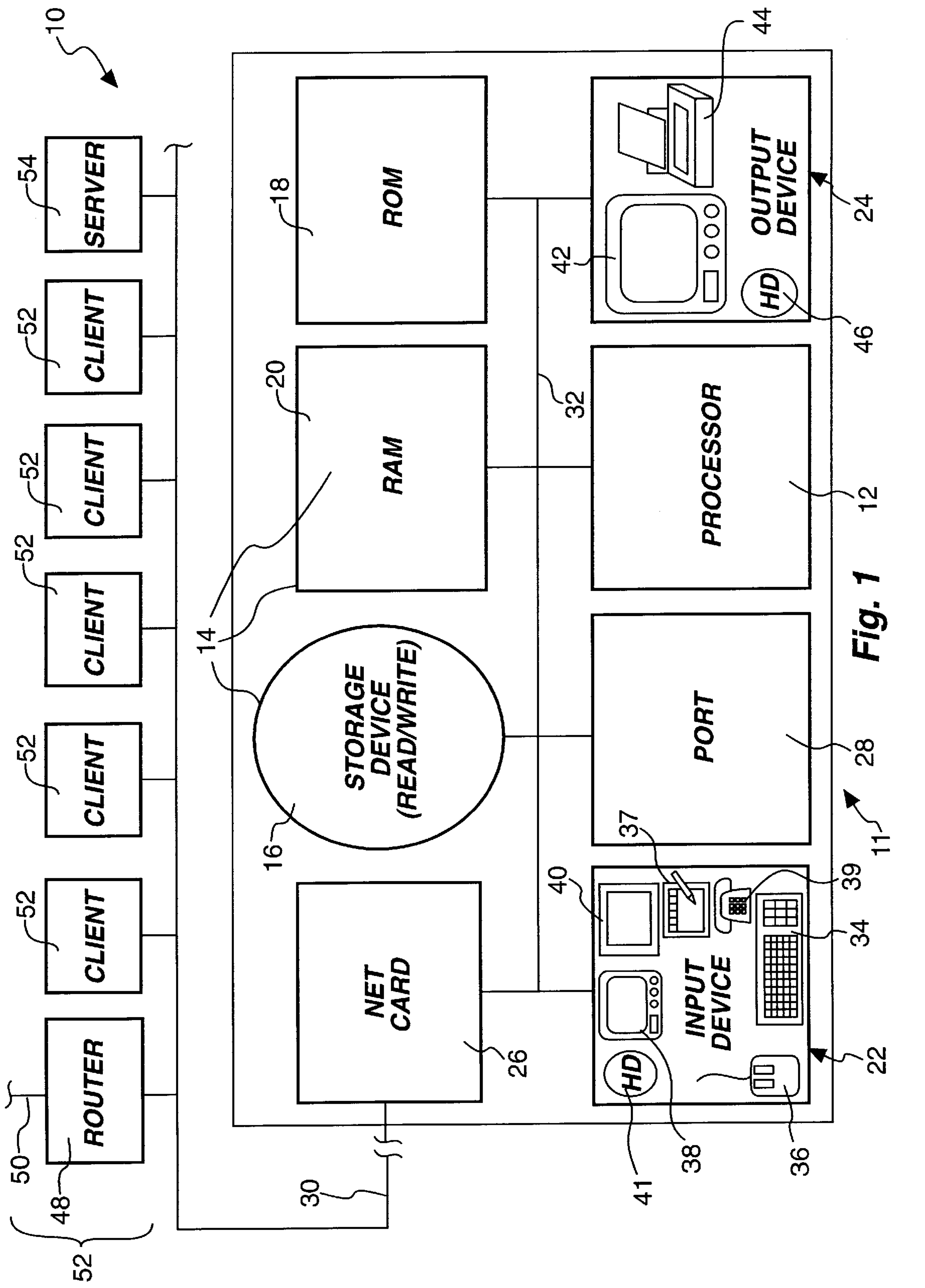
Primary Examiner—Norman M. Wright
(74) *Attorney, Agent, or Firm*—Pate Pieree & Baird

(57) ABSTRACT

A system and method for providing detection of the signatures effected by a defective Floppy Diskette Controller ("FDC") operates on media independent of files thereon, or on files, independent of the media on which they are stored. Multiple testing strategies incorporate evaluations to detect signatures of data corruption introduced by defective FDCs from long transfer delays, short transfer delays, contiguous storage of logical sectors, or fragmented storage of logical sectors of a file. A false positive filter uses secondary testing of data. Filters remove from consideration those common patterns that properly and naturally occur. These filters rely on indicia demonstrating that primary leading indicators of the presence of an error do not really result from an actual error. The signatures may be detected regardless of subsequent transfer of corrupted files to various media including the media tested.

26 Claims, 7 Drawing Sheets





10
SERVER
CLIENT
CLIENT
CLIENT
CLIENT
CLIENT
ROUTER
54
52
52
52
52
52
48
50
52
30
ROM
RAM
STORAGE DEVICE (READ/WRITE)
NET CARD
18
20
14
16
26
OUTPUT DEVICE
PROCESSOR
PORT
INPUT DEVICE
HD
HD
44
42
46
24
32
12
28
11
37
40
39
34
22
41
38
36

Fig. 1

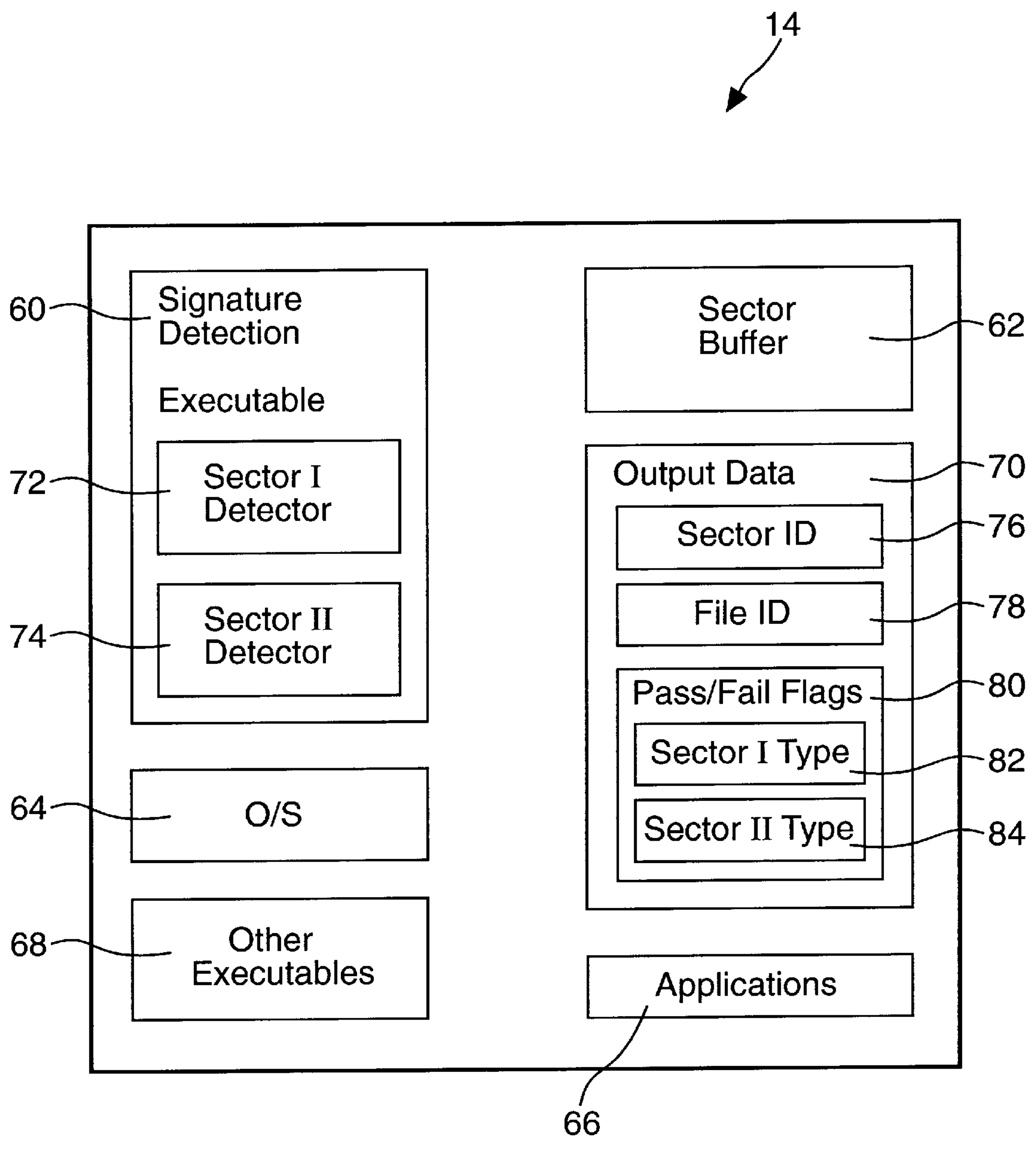

14
60
Signature Detection
Executable
72
Sector I Detector
74
Sector II Detector
64
O/S
68
Other Executables
Sector Buffer
62
Output Data
70
Sector ID
76
File ID
78
Pass/Fail Flags
80
Sector I Type
82
Sector II Type
84
Applications
66

Fig. 2

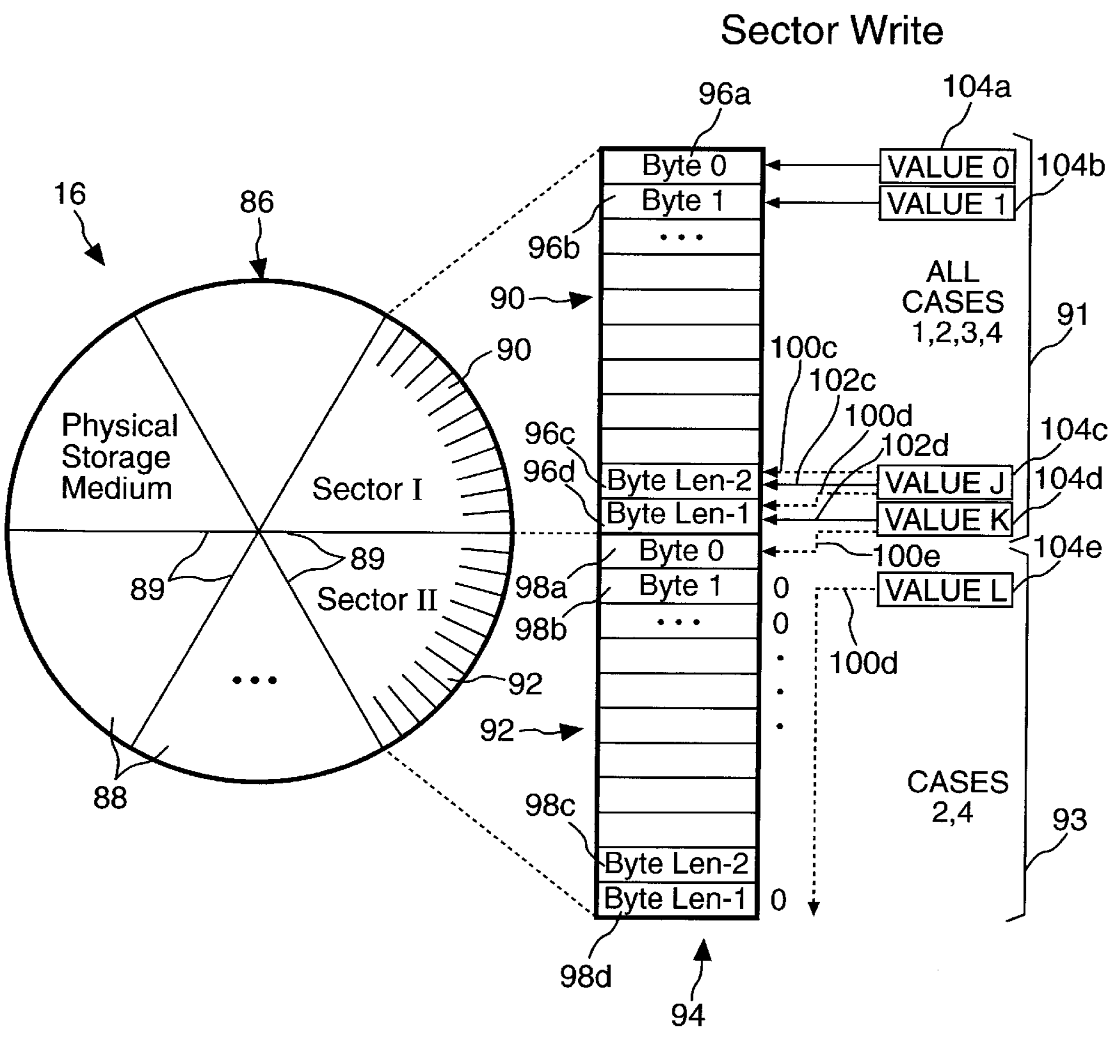

Sector Write
96a
104a
16
86
Byte 0
Byte 1
• • •
VALUE 0
VALUE 1
104b
96b
90
ALL CASES 1,2,3,4
91
90
100c
102c
100d
102d
104c
104d
Physical Storage Medium
Sector I
96c
96d
Byte Len-2
Byte Len-1
VALUE J
VALUE K
Byte 0
Byte 1
100e
104e
89
89
Sector II
98a
98b
0
0
VALUE L
100d
92
92
• • •
88
98c
CASES 2,4
93
Byte Len-2
Byte Len-1
0
98d
94

Fig. 3

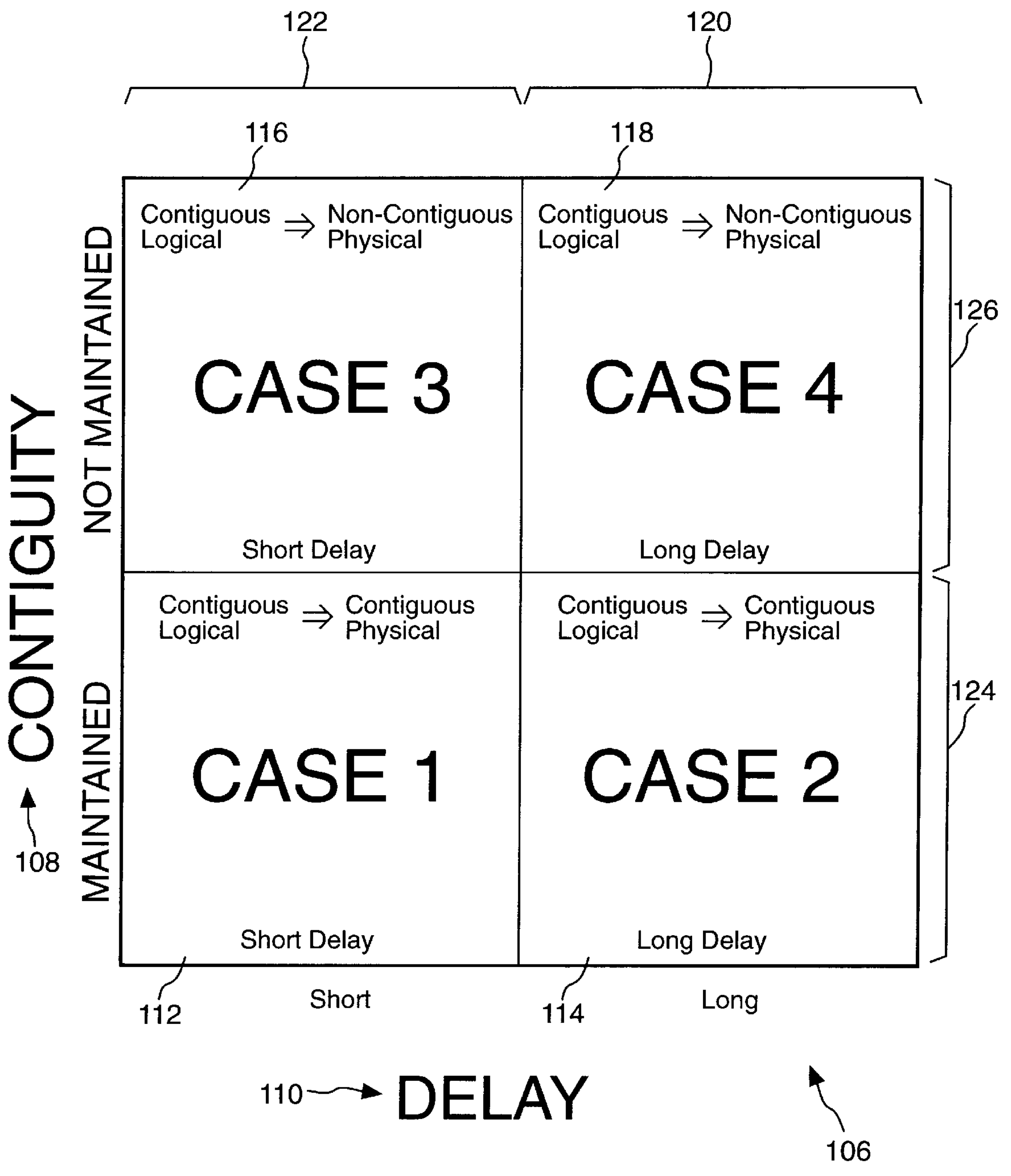

122
120
116
118
Contiguous Logical ⇒ Non-Contiguous Physical
CASE 3
Short Delay
Contiguous Logical ⇒ Non-Contiguous Physical
CASE 4
Long Delay
126
Contiguous Logical ⇒ Contiguous Physical
CASE 1
Short Delay
Contiguous Logical ⇒ Contiguous Physical
CASE 2
Long Delay
124
CONTIGUITY
NOT MAINTAINED
MAINTAINED
108
112
Short
114
Long
110
DELAY
106

Fig. 4

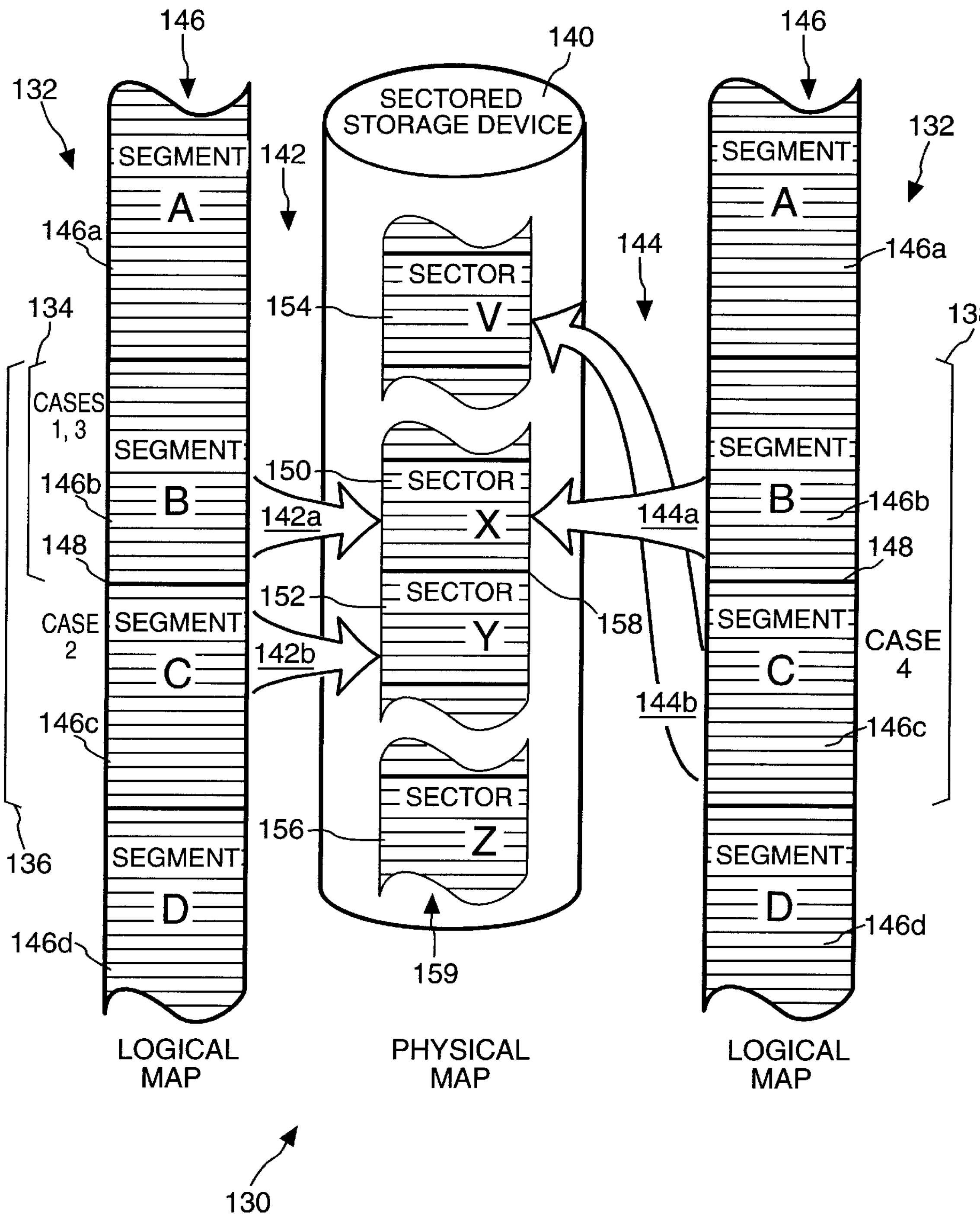

146
132
SEGMENT
A
146a
134
CASES 1, 3
SEGMENT
B
146b
148
CASE 2
SEGMENT
C
146c
136
SEGMENT
D
146d
LOGICAL MAP
140
SECTORED STORAGE DEVICE
142
154
SECTOR
V
150
142a
SECTOR
X
152
SECTOR
Y
142b
156
SECTOR
Z
159
PHYSICAL MAP
144
144a
158
144b
146
132
SEGMENT
A
146a
138
SEGMENT
B
146b
148
SEGMENT
C
CASE 4
146c
SEGMENT
D
146d
LOGICAL MAP
130

Fig. 5

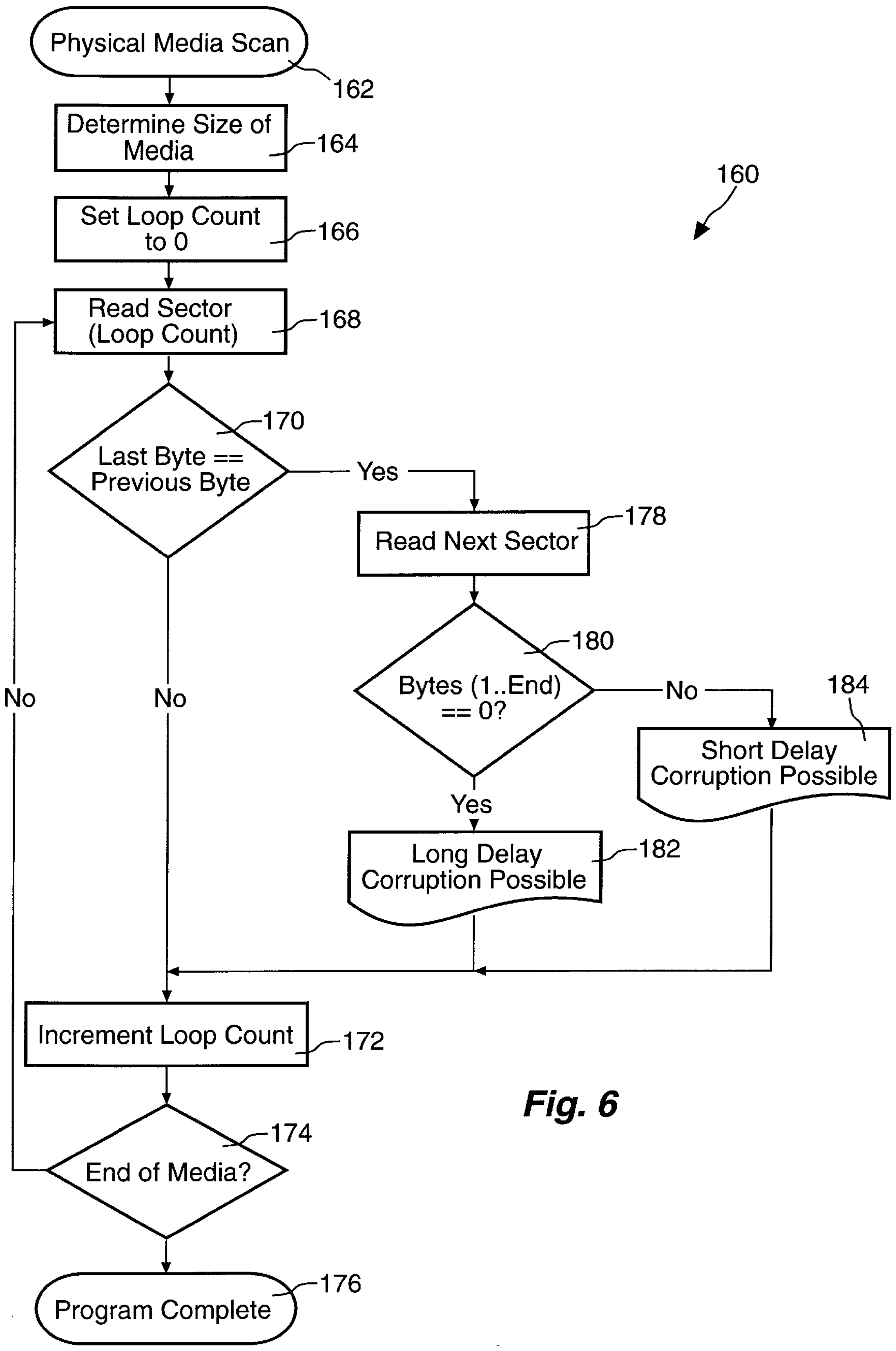
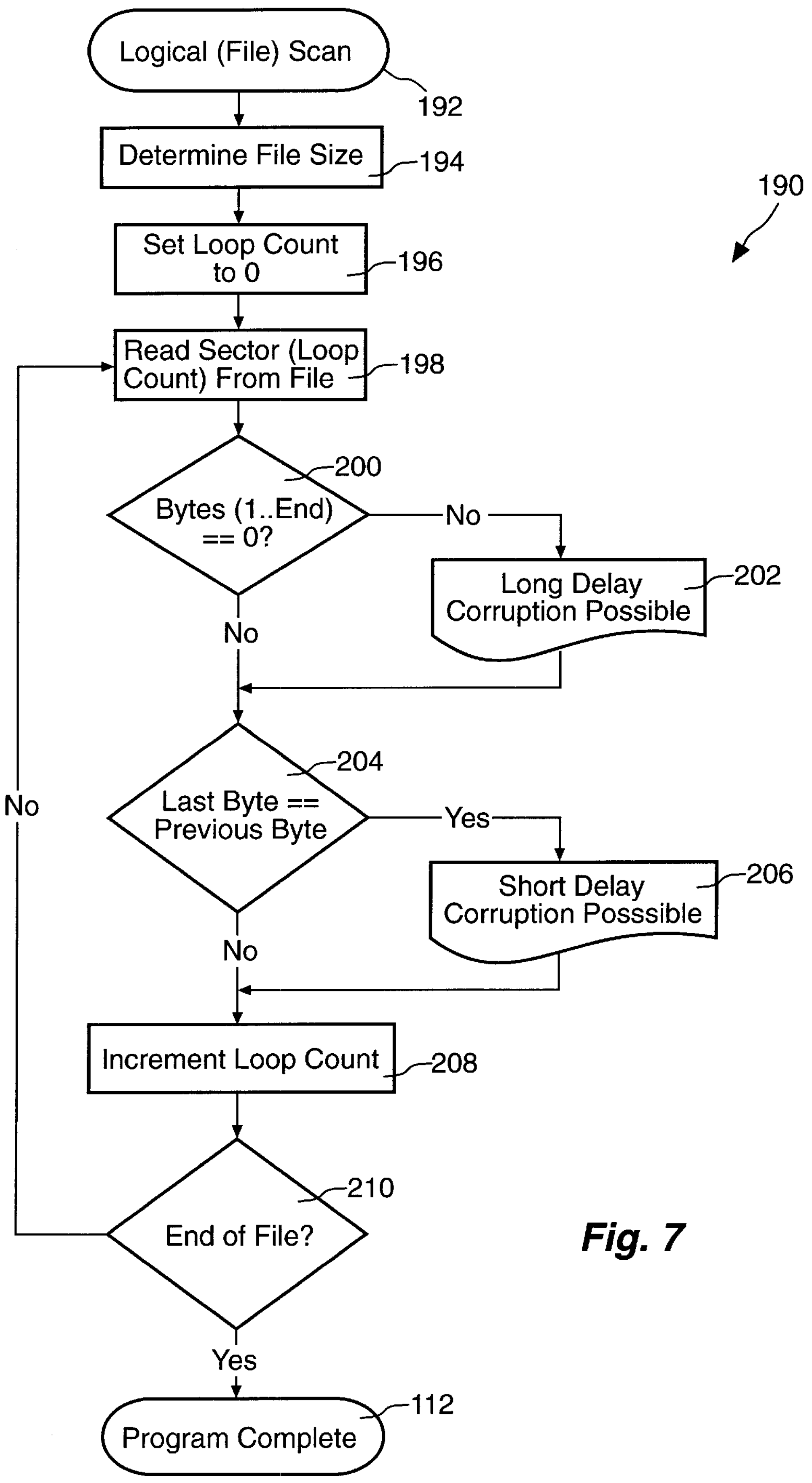

Physical Media Scan
162
Determine Size of Media
164
Set Loop Count to 0
166
Read Sector (Loop Count)
168
Last Byte == Previous Byte
170
Yes
Read Next Sector
178
Bytes (1..End) == 0?
180
No
184
Short Delay Corruption Possible
Yes
Long Delay Corruption Possible
182
No
No
Increment Loop Count
172
End of Media?
174
Program Complete
176
160

Fig. 6



Logical (File) Scan
192
Determine File Size
194
190
Set Loop Count to 0
196
Read Sector (Loop Count) From File
198
200
Bytes (1..End) == 0?
No
Long Delay Corruption Possible
202
No
204
Last Byte == Previous Byte
Yes
Short Delay Corruption Posssible
206
No
No
Increment Loop Count
208
210
End of File?
Yes
Program Complete
112

Fig. 7

DATA CORRUPTION DETECTION APPARATUS AND METHOD

BACKGROUND

1. The Field of the Invention

This invention relates to the detection of corruption occurring in data written to storage media relying on a defective Floppy Diskette Controller ("FDC"), where an undetected data error causes data corruption and, more particularly, to novel systems and methods for inspection and warning to enable prompt restoration of data corrupted by defective FDCs.

2. The Background Art

Computers are now used to perform functions and maintain data critical to many organizations. Businesses use computers to maintain essential financial and other business data. Computers are also used by government to monitor, regulate, and even activate, national defense systems. Maintaining the integrity of the stored data is essential to the proper functioning of these computer systems, and data corruption can have serious (even life threatening) consequences.

Most of these computer systems include diskette drives for storing and retrieving data on floppy diskettes. For example, an employee of a large financial institution might have a personal computer that is attached to the main system. In order to avoid processing delays on the mainframe, the employee may routinely transfer data files from the host system to his local personal computer and then back again, temporarily storing data on a local floppy diskette. Similarly, an employee with a personal computer at home may occasionally decide to take work home, transporting data away from and back to the office on a floppy diskette.

Data transfer to and from a floppy diskette is controlled by a device called a Floppy Diskette Controller ("FDC"). The FDC is responsible for interfacing the computer's Central Processing Unit ("CPU") with the physical diskette drive. Significantly, since the diskette is spinning, it is necessary for the FDC to provide data to the diskette drive at a specified data rate. Otherwise, the data will be written to the wrong location on the diskette.

The design of the FDC accounts for situations when the data rate is not adequate to support the rotating diskette. Whenever this situation occurs, the FDC aborts the operation and signals the CPU that a data underrun condition has occurred. Unfortunately, however, it has been found that a design flaw in many FDCs makes it impossible to detect all data underrun conditions. This flaw has, for example, been found in the NEC 765, INTEL 8272 and compatible Floppy Diskette Controllers. Specifically, data loss and/or data corruption can occur during data transfers to or from diskettes (or even tape drives and other media which employ the FDC), whenever the last data byte of a sector being transferred is delayed for more than a few microseconds. Furthermore, if the last byte of a sector write operation is delayed too long then the next (physically adjacent) sector of the diskette will be destroyed as well.

For example, it has been found that these FDCs cannot detect a data underrun on the last byte of a diskette read or write operation. Consequently, if the FDC is preempted during a data transfer to the diskette (thereby delaying the transfer), and an underrun occurs on the last byte of a sector, the following occurs: (1) the underrun flag does not get set, (2) the last byte written to the diskette is made equal to the

previous byte written, and (3) Cyclic Redundancy Check ("CRC") is generated on the altered data. The result is that incorrect data is written to the diskette and validated by the FDC.

Conditions under which this problem may occur can be identified by simply identifying those conditions that can delay data transfer to or from the diskette drive. In general, this requires that the computer system be engaged in "multi-tasking" operation or in overlapped input/output ("I/O") operation. Multi-tasking is the ability of a computer operating system to simulate the concurrent execution of multiple tasks. Importantly, concurrent execution is only "simulated" because there is usually only one CPU in today's personal computers, and it can only process one task at a time. Therefore, a system interrupt is used to rapidly switch between the multiple tasks, giving the overall appearance of concurrent execution.

MS-DOS and PC-DOS, for example, are single-task operating systems. Therefore, one could argue that the problem described above would not occur. However, there are a number of standard MS-DOS and PC-DOS operating environments that simulate multi-tasking and are susceptible to the problem. The following environments, for example, have been found to be prime candidates for data loss and/or data corruption due to defective FDCs: local area networks, 327x host connections, high density diskettes, control print screen operations, terminate and stay resident ("TSR") programs. The problem has also been found to occur as a result of virtually any interrupt service routine. Thus, unless the MS-DOS and PC-DOS operating systems disable all interrupts during diskette transfers, they are also susceptible to data loss and/or corruption.

The UNIX operating system is a multi-tasking operating system, and it is extremely simple to create a situation that can cause the problem within UNIX. One of the more simple examples is to begin a large transfer to the diskette and place that task in the background. After the transfer has begun then begin to process the contents of a very large file in a way that requires the use of a higher-priority Direct Memory Access ("DMA") channel than the floppy diskette controller's DMA channel, i.e., video updates, multi-media activity, etc. Video access forces the video buffer memory refresh logic on DMA channel **1**, along with the video memory access, which preempts the FDC operations from occurring on DMA channel **2** (which is lower priority than DMA channel **1**). This type of example creates the classic overlapped I/O environment and can force the FDC into an undetectable error condition. More rigorous examples could include the concurrent transfer of data to or from a network or tape drive using a high priority DMA channel while the diskette transfer is active. Clearly, the number of possible error producing examples is infinite and very possible in this environment.

For all practical purposes the OS/2 and newer Windows operating systems can be regarded as UNIX derivatives. In other words, they suffer from the same problems that UNIX does. There are, however, two significant differences between these operating systems and UNIX. First, they both semaphore video updates with diskette operations in an effort to avoid forcing the FDC problem to occur. However, any direct access to the video buffer, in either real or protected mode, during a diskette transfer will bypass this safe-guard and result in the same condition as UNIX. Second, OS/2 incorporates a unique command that attempts to avoid the FDC problem by reading back every sector that is written to the floppy diskette in order to verify that the operation completed successfully. This command is an extension to the MODE command (MODE DSKT VER=ON). With these changes, data loss and/or data corruption should occur less frequently than before, but it is still possible for the FDC problem to destroy data that is not related to the current sector operation.

There are a host of other operating systems that are susceptible to the FDC problem just like DOS, Windows, Windows 95, Windows NT, OS/2, and UNIX. However, these systems may not have an installed base as large as DOS, Windows, OS/2 or UNIX, and there may, therefore, be little emphasis on addressing the problem. Significantly, as long as the operating systems utilize the FDC and service system interrupts, the problem can manifest itself. This can, of course, occur in computer systems which use virtually any operating system.

Some in the computer industry have suggested that the FDC problem is extremely rare and difficult to reproduce. This is similar to the argument presented during the 1994 defective INTEL Pentium scenario. Error rates for the defective Pentium ranged from microseconds to tens-of-thousands of years! Admittedly, the FDC problem is often very difficult to detect during normal operation because of its random characteristics. The only way to visibly detect this problem is to have the FDC corrupt data that is critical to the operation at hand. There may, however, be many locations on the diskette that have been corrupted, but not accessed. Studies have recently demonstrated that the FDC problem is quite easy to reproduce and may be more common than heretofore believed.

Computer users may, in fact, experience this problem frequently and not even know about it. After formatting a diskette, for example, the system may inform the user that the diskette is bad, although the user finds that if the operation is performed again on the same diskette everything is fine. Similarly, a copied file may be unusable, and the computer user concludes that he or she just did something wrong. For many in this high-tech world, it is very difficult to believe that the machine is in error and not themselves. It remains a fact, however, that full diskette back-ups are seldom restored, that all instructions in programs are seldom, if ever, executed, that diskette files seldom utilize all of the allocated space, and that less complex systems are less likely to exhibit the problem.

Additionally, the first of these FDCs were shipped in the late 70's. The devices were primarily used at that time in special-purpose operations in which the FDC problem would not normally be manifest. Today, on the other hand, the FDCs are incorporated into general-purpose computer systems that are capable of concurrent operation (multi-tasking or overlapped I/O). Thus, it is within today's environments that the problem is most likely to occur by having one of the operations delay the data transfer to the diskette. The more complex the computer system, the more likely it is to have one activity delay another, thereby creating the FDC error condition.

In short, the potential for data loss and/or data corruption is present in all computer systems that utilize the defective version of this type of FDC, presently estimated at about 25 million personal computers. The identification and repair of defective FDCs has been described in previously filed U. S. patent applications.

In addition to a solution to the FDC problem it is necessary to be able to accurately, and correctly, identify defective, corrupted data before that data is relied upon at great loss. The design flaw in the FDC causes data corruption to occur and manifest itself in the same manner as a

destructive computer virus. Furthermore, because of its nature, this problem has the potential of rendering even secure databases absolutely useless.

The defect in FDCs, however, results in various types of corruption having different signatures, according to the nature of the defective FDC and the nature of the conditions at transfer. Moreover, files may be transferred, fragmented, defragmented, and the like many times over years. Thus, corruption may be spread and the corrupted files relied upon at any time. Locating the possibility or probability of corruption in an individual file or a sector of storage media is a first step toward restoring reliable information before corruption can cause serious harm.

The aforementioned delay (long or short) in a transfer of a last data byte of a sector either to or from a floppy diskette at any time in the history of a file, may cause corruption. The length of the transfer delay may alter the nature of the corruption which corruption may then be copied or transferred any number of times before being relied upon.

Files may also be fragmented or defragmented. Accordingly, a logical file may be written to contiguous or non-contiguous sectors of any particular medium. Transfers, fragmentation, de-fragmentation, and the like may occur long after an initial occurrence of corruption, further obscuring the more obvious signatures of corruption.

Thus, an apparatus and method are needed to detect the possibility of corruption from either long or short delays in transfers controlled by defective FDCs. File integrity must be tested regardless of fragmentation (non-contiguity) of sectors holding logically consecutive data. Integrity must be testable whether subsequent transfers have occurred by any means which may or may not have affected logical or physical contiguity of sectors' contents.

BRIEF SUMMARY AND OBJECTS OF THE INVENTION

In view of the foregoing, it is a primary object of the present invention to provide a system and method for the detection of data corruption due to defective Floppy Diskette Controllers ("FDCs").

It is also an object of the present invention to provide an automated software-only (programmatic) approach to reduce the labor of searching files and media.

Consistent with the foregoing objects, and in accordance with the invention as embodied and broadly described herein, a system and method are disclosed in one embodiment of the present invention as including a detection program that is capable of correctly and accurately detecting the signature of data corruption associated with defective FDCs.

The approach taken includes testing according to physical media configuration, and according to logical file configuration. The tests report the presence of any of the signatures known to be associated with defective FDCs. The system manager or other responsible party can then restore the files from an uncorrupted archival copy, if available. In any event, A warning may thus be available to identify individual files as well as sectors of media that are not reliable.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects and features of the present invention will become more fully apparent from the following description and appended claims, taken in conjunction with the accompanying drawings. Understanding that these drawings depict only typical embodiments of the invention and are, therefore, not to be considered limiting of its scope, the invention will be described with additional specificity and detail through use of the accompanying drawings in which:

FIG. **1** is a schematic block diagram of a system consistent with a computer hosting executables and data to implement the invention;

FIG. **2** is a schematic block diagram of data structures containing executables and operational data for implementing an embodiment of the invention on the apparatus of FIG. **1**;

FIG. **3** is a schematic block diagram illustrating a physical view of data storage on a sectored storage medium;

FIG. **4** is a schematic block diagram illustrating the combinations of conditions that may create corruption detectable by an apparatus and method in accordance with the invention;

FIG. **5** is a schematic block diagram of a sectored storage device and its relationship to logical maps of files that may be stored thereon before or after corruption of sectors;

FIG. **6** is a schematic block diagram illustrating a method for detecting corruption by scanning physical storage media; and

FIG. **7** is a schematic block diagram of a method for detecting corruption caused by defective floppy diskette controllers by scanning files according to logical structure.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

It will be readily understood that the components of the present invention, as generally described and illustrated in the Figures herein, could be arranged and designed in a wide variety of different configurations. Thus, the following more detailed description of the embodiments of the system and method of the present invention, as represented in FIGS. **1** through **7**, is not intended to limit the scope of the invention, as claimed, but it is merely representative of the presently preferred embodiments of the invention.

The presently preferred embodiments of the invention will be best understood by reference to the drawings, wherein like parts are designated by like numerals throughout.

Referring to FIG. **1**, an apparatus **10** may implement the invention on one or more nodes **11**, (client **11**, computer **11**) containing a processor **12** (CPU **12**). All components may exist in a single node **11** or may exist in multiple nodes **11**, **52** remote from one another. The CPU **12** may be operably connected to a memory device **14**. A memory device **14** may include one or more devices such as a hard drive or other non-volatile storage device **16**, a read-only memory **18** (ROM) and a random access (and usually volatile) memory **20** (RAM/operational memory).

The apparatus **10** may include an input device **22** for receiving inputs from a user or another device. Similarly, an output device **24** may be provided within the node **11**, or accessible within the apparatus **10**. A network card **26** (interface card) or port **28** may be provided for connecting to outside devices, such as the network **30**.

Internally, a bus **32** may operably interconnect the processor **12**, memory devices **14**, input devices **22**, output devices **24**, network card **26** and port **28**. The bus **32** may be thought of as a data carrier. As such, the bus **32** may be embodied in numerous configurations. Wire, fiber optic line, wireless electromagnetic communications by visible light, infrared, and radio frequencies may likewise be implemented as appropriate for the bus **32** and the network **30**.

Input devices **22** may include one or more physical embodiments. For example, a keyboard **34** may be used for interaction with the user, as may a mouse **36** or stylus pad. A touch screen **38**, a telephone **39**, or simply a telephone line **39**, may be used for communication with other devices, with a user, or the like. Similarly, a scanner **40** may be used to receive graphical inputs which may or may not be translated to other character formats. The hard drive **41** or other memory device **41** may be used as an input device whether resident within the node **11** or some other node **52** (e.g., **52***a,* **52***b,* etc.) on the network **30**, or from another network **50**.

Output devices **24** may likewise include one or more physical hardware units. For example, in general, the port **28** maybe used to accept inputs and send outputs from the node **11**. Nevertheless, a monitor **42** may provide outputs to a user for feedback during a process, or for assisting two-way communication between the processor **12** and a user. A printer **44** or a hard drive **46** may be used for outputting information as output devices **24**.

In general, a network **30** to which a node **11** connects may, in turn, be connected through a router **48** to another network **50**. In general, two nodes **11**, **52** may be on a network **30**, adjoining networks **30**, **50**, or may be separated by multiple routers **48** and multiple networks **50** as individual nodes **11**, **52** on an internetwork. The individual nodes **52** (e.g. **11**, **52**, **54**) may have various communication capabilities.

In certain embodiments, a minimum of logical capability may be available in any node **52**. Note that any of the individual nodes **11**, **52**, **54** may be referred to, as may all together, as a node **11** or a node **52**. Each may contain a processor **12** with more or less of the other components **14**–**44**.

A network **30** may include one or more servers **54**. Servers may be used to manage, store, communicate, transfer, access, update, and the like, any practical number of files, databases, or the like for other nodes **52** on a network **30**. Typically, a server **54** may be accessed by all nodes **11**, **52** on a network **30**. Nevertheless, other special functions, including communications, applications, directory services, and the like, may be implemented by an individual server **54** or multiple servers **54**.

In general, a node **11** may need to communicate over a network **30** with a server **54**, a router **48**, or nodes **52**. Similarly, a node **11** may need to communicate over another network (**50**) in an internetwork connection with some remote node **52**. Likewise, individual components **12**–**46** may need to communicate data with one another. A communication link may exist, in general, between any pair of devices.

Referring to FIG. **2**, a storage device **14**, may be loaded with executables and data. For execution, of the storage device **14** may be the RAM **20**. For initial installation, the memory device **14** selected may be another storage device **16** or ROM **18**. In general, executables and operational data ready to be executed by a processor **12** may implemented in a memory device **14** corresponding to RAM **20**.

A signature detection executable **60** may contain instructions in suitable code for implementing algorithms. The signature detection executable **60** may operate with sector buffer **62**. The sector buffer **62** is sized to store data select for evaluation. Evaluation, conducted by the signature detection executable **60** includes analysis of the contents of data stored on media to be tested. In one embodiment, the sector buffer **62** may include one or more buffers **62**. Alternatively, the sector buffer **62** may include sufficient space to hold at least two complete sectors from a storage medium to be tested.

A processor **12** requires some underlying operating system **64** in order to run the executable **60**. Similarly, applications **66** and other executables **68** may be hosted in the memory device **14**. In one presently preferred embodiment, the memory device **14** is the random access memory (RAM) **20** of FIG. **1**.

Output data **70** may be stored during operation of the signature detection executable **60** on the processor **12**. The output data **70** indicates the nature of any corruption signature found by the signature detection executable **60**.

The signature detection executable **60** may include detector **72** for distinguishing corruption peculiar to a primary, leading, or first sector. A detector **74** may be programmed to identify corruption normally associated with a following, secondary, or second sector involved in corruption by defective floppy diskette controller (FDC). Each of the detectors **72**, **74** may be programmed to operate on a logical basis or physical basis. That is, in one embodiment, an apparatus and method in accordance with the invention may operate based on files. Accordingly, the file system associated with a computer **11** may be relied upon to define the location of an initial sector, subsequent sectors, and a final sector associated with a single file at a time. Thus, regardless of the random nature of storage on any storage device **14**, a file may be tested for integrity.

Similarly, a detector **72**, **74** may be programmed to operate on any particular storage medium **16**, **18**, **20**. For example, a storage device **16** may be a floppy diskette or a hard drive. The ROM **18** may be configured in a chip, or on a laser-readable compact disk. In certain embodiments, the detectors **72**, **74** may scan and evaluate the entire medium within a particular memory device **14**. Thus, any physical sectors containing the signature identified with corruption by a defective floppy diskette controller may be detected, regardless of subsequent transfer to any other storage device **14**. Since storage is typically done on a sector basis, corrupted sectors may be detected over an entire storage medium, or over a particular file on a storage medium.

The output data **70** may include any information deemed suitable to enable ready identification of files, responsible individuals, and the like. Perhaps most importantly, the output data **70** may include information identifying files, and personnel responsible for those files, in order to enable prompt restoration of corrupted files.

In one embodiment, the output data **70** may include sector identification **76**. Sector identification **76** (sector ID **76**) may include not only a sector number, but a volume number, a network address of a computer **11** on which the defective sector is located, and the like. Thus, an entire path may identify a sector by any path or context required.

A file identifier **78** (file ID **78**) may identify a particular file in which corruption is detected. A file system will typically contain a file name as well as higher level path identification associated with a user, computer **11**, volume, directory, and the like. A file identifier **78** may include any information deemed suitable to rapidly and effectively single out a file containing corruption. Likewise, sufficient context may be provided in the file ID **78** to enable a user to locate a source of such corrupted file. Accordingly, a user, system manager, or other responsible party may be able to more rapidly identify a source file from which a corrupted file may be restored. Likewise, a source file may be corrupted. Accordingly, identification of a file with sufficient detail to identify its source may provide identification of other storage media to be tested for corruption.

In one embodiment, pass fail flags **80** may be included as output data **70**. For example, in one embodiment, every sector in a storage medium may be identified as passing or failing a test in accordance with the invention. Similarly, every file in a volume or a server may be tested and identified as having passed or failed a test in accordance with the invention. However, in one currently preferred embodiment of an apparatus and method in accordance with the invention, only sectors of a medium or a file displaying a corruption signature need be identified. Thus, the nature of such corruption signature may be identified. For example, corruption occurs in a primary sector due to improper writing and error checking by a defective floppy diskette controller. Depending on the length of a delay, the corruption may extend to a subsequent sector. Thus, a sector I type flag **82** may identify a sector as containing corruption on the type identified by a sector I detector **72**. Similarly, a sector II type flag **84** may identify a sector as containing corruption having the signature detected by the sector II detector **74**.

Referring to FIG. **3**, specifically, and to FIGS. **2**–**5**, generally, a storage device **16** may include a storage medium **86**. The storage medium **86** may contain one or more disks or diskettes. In general, data corruption may be initiated by a defective floppy diskette controller on a particular diskette. However, in general, a file or sector thus corrupted may be copied to any other memory device **14**. Thus, a storage device **16** being tested for corruption may be a diskette, a hard disk, or other storage device **14** to which data may have been transferred subsequent to storage on a floppy diskette.

The storage medium **86** may contain sectors **88**, subdivisions **88** into which medium **86** may be subdivided for purposes of addressing and segmenting data. The sectors **88** may be separated by sector boundaries **89** specified in a formatting standard used to format the storage medium **86**. For convenience, a sector I **90** and sector II **92** are identified. Each of the sectors **90**, **92** may be physically represented by a map **94** of individual bytes. The number of bytes in a particular sector **90**, **92** is established by an appropriate standard. Thus, a first byte **96*a*** in sector I **90** has a number of zero. The second byte **96*b*** is identified as byte one. Thus, a last byte **96*d*** is a byte identified by the length of the sector **90**, less one. Likewise, the next-to-last byte **96*c*** is counted according to a length, less two, of the sector **90**. In sector II **92**, a first byte **98*a*,** second byte **98*b*,** next-to-last byte **98*c*,** and last byte **98*d*,** may be thought of as similarly numbered.

In FIG. **3**, the paths **100** illustrate the effect of a defective floppy diskette controller, under corrupting conditions. The paths **102** illustrate the paths that particular bytes **96**, **98** should take in a non-defective floppy diskette controller, or in a defective floppy diskette controller under non-corrupting conditions.

Various values **104** may be placed in the byte locations **96**, **98** in the sectors **90**, **92**, respectively. For example, a value zero **104*a*** is stored to the byte zero location **96*a*.** A value one **104*b*** is stored to the byte one location **96*b*.**

In normal operation, a value J **104*c*** is stored at the next-to-last byte location **96*c*,** while a value K **104*d*** is stored to a last byte location **96*d*.** Sector II **92** should remain unaffected by the transfer of values **104** to sector I **90**.

In all cases of data corruption due to defective FDC's, the value J **104*c*** intended for the next-to-last byte location **96*c*** is stored at the proper location **96*c*.** Thus, the intended path **102*c*** for normal operation is duplicated by the path **100*c*** when corruption is incipient.

However, the value J **104*c*** in the presence of the corrupting conditions for a defective FDC, is transferred along the path **100*d*** to the last byte location **96*d*.** Normally, the value K **104*d*** that would be transferred along the path **102*d*** to a last byte location **96*d*** is detoured.

The value K **104***d* passes along the path **100***e* to the first byte location **98***a* in sector II **92**. Thus, the last byte location **96***d* contains the same value J **104***c* that is written to the next-to-last byte location **96***c*. Meanwhile, a value L **104***e*, having an actual numerical value of zero, is written to all other byte locations **98***b*, **98***c*, **98***d* up to the last byte location **98***e* of sector II **92**.

One fundamental cause of corruption is delay in writing a value K **104***d* to a last byte location **96***d*. If the delay is greater than a single byte write time (32 μs or 16 μs) and less than 80 microseconds, the delay is considered to be a "short delay." If the delay is greater than 80 microseconds, then a "long delay" has occurred. If a short delay occurs, then the value K **104***d* is not written to the last byte location **96***d*, nor anywhere else. However, if the delay is long, then sector II **92** will be effected.

The mapping of values **104** to byte locations **94** in FIG. **3**, in accordance with the normal paths **102** and the corruption paths **100** varies according to certain conditions or cases. FIG. **4** illustrates the conditions and cases that various scenarios may present with a defective floppy diskette controller.

Referring to FIG. **4**, a matrix **106** relates a contiguity **108** and delay **110** to create various cases **112**, **114**, **116**, **118**. Contiguity **108** refers to whether or not a file has been fragmented or defragmented. For example, a file has a logical flow. Nevertheless, the data corresponding to a particular file may be stored in randomly distributed sectors **88** within a storage medium **86**. Contiguity of adjacent sectors **90**, **92** may maintained. Alternatively, contiguity **108** may also not be maintained. Similarly, a delay greater than 80 microseconds is considered a long delay **120**. A delay of less 80 microseconds is considered a short delay **122**. As discussed above, the corruption signature varies according to whether or not the delay **110** in a transfer of the values **104** to a sector **88** is controlled according to the length of the delay **110**.

Case **1** corresponds to a short delay **122** in transferring values **104** to byte locations **96**. In case **1 112** also corresponds to maintained contiguity **124**. Contiguity **108** may be maintained **124** or not maintained **126**. FIG. **4** illustrates maintenance **124** and non-maintenance **126** of contiguity **108**. Contiguity **108** refers to the writing of logically contiguous data onto physically contiguous sectors **90**, **92**.

Case **1 112** has conditions of a short delay **122** and maintained contiguity **124**. Since the delay **110** is short **122**, only a sector I **90** is affected.

Case **2 114** has conditions corresponding to a long delay **120** and maintained contiguity **124**. Since case **2 114** includes a long delay **120**, corruption may occur in both sector I **90**, and sector II **92** of the same logical file unless sector I **90** is the last sector of the file thus causing corruption in logically unrelated locations.

Case **3 116** has conditions corresponding to a short delay **122** and non-maintained contiguity **126**. Because the delay **110** is short **122**, case **3 116** may result in corruption only in sector I **90**. Sector II **92** remains unaffected.

Case **4 118** includes corresponding conditions of a long delay **120** and non-contiguity **126**. The long delay **120** can cause corruption to occur in both sector I **90** and sector II **92**. Sector II **92** is not logically related to sector I **90** potentially causing data corruption to another (unrelated) file. One may note that the delay **110**, whether long **120** or short **122**, appears to control the presence of corruption in sector II **92**. Contiguity **108** does not appear to be a factor in the nature of the corruption.

Contiguity **108** or maintenance **124** and non-maintenance **126** of contiguity **108** does not control the presence of corruption, but rather the signature thereof. Thus, FIGS. **3**–**5** should be viewed together.

Referring to FIG. **5**, a map **130** of the file **132** is illustrated under various sets **134**, **136**, **138** of conditions, or simply under scenarios **134**, **136**, **138** or conditions **134**, **136**, **138**. The set **134** corresponds to conditions of case **1 112** and case **3 116**. The conditions **136** or set **136** of conditions, corresponds to case **2 114** in FIG. **4**. The set **138** of conditions, or condition **138** corresponds to case **4 118** in FIG. **4**. The conditions **134** or set **134** corresponds to a short delay **122**, and thus a short delay corruption signature **91** or sector I corruption **91**. The case **136** or set **136** corresponds to sector II corruption **93** or long delay corruption **93** with maintained contiguity **124** between a logical map **132** and a physical map **140**. Meanwhile, the conditions **138** or set **138** corresponds to long delay corruption **93** or sector II corruption **93** corresponding to a long delay **120** wherein contiguity **108** is not maintained **126**.

A storage device **140** is sectored to receive data. Data may be transferred **142** continually (maintaining contiguity **124**). Data may also be transferred from **144** with contiguity **108** not maintained **126**. Note that trailing alphabetical characters after reference numerals merely identify instances of the principle or generic feature identified by the reference numeral.

For example, a file **132** may be divided into segments **146**. The segments **146***a*, **146***b*, **146***c*, **146***d* are illustrated in a sequential, logical, and contiguous arrangement. Segments **146** may correspond to sector-sized portions of a file **132** or logical map **132** of data or code. The individual segments **146** may be thought of as being divided at segment boundaries **148**.

Similarly, the storage device **140** may be sectored into individual sectors **150**, **152**, **154**, **156**, separated by sector boundaries **158**. The sectors may also be referred to generically as sectors **159**, or as a sector **159**. Notice that sectors **150**, **152** are illustrated schematically as being contiguous. Sectors **154**, **156** may be separated from the sectors **150**, **152** by some other number of individual sectors **159**.

Under the set **134** of conditions, a segment **146***b* may be transferred **142***a* to a sector **150**. Under the conditions **134**, corresponding to case **1 112** and case **3 116**, said transfer **142***a* does not affect the subsequent segment **146***c*, nor the subsequent sector **152**. Rather the transfer **142***b* occurs without an influence of the corruption that may be included in a transfer **142***a*. This condition corresponds to a short delay **122**. Thus, a corruption signature in the sector **150** will include a value J **104***c* in a next-to-last byte location **96***c* in the sector **150**. Likewise, a value J **100***d* will be stored in the last byte location **96***d* of sector **150** (see FIG. **3**). Because the delay **110** is a short delay **122**, the value K **104***d* that should have been transferred along the path **102***d* to the last byte location **96***d* is simply lost. The value K **104***d* is not written to the subsequent sector **152** along the path **100***e*. The conditions **136** corresponding to case **2 114**, include the conditions **134** of case **1 112** and case **3 116**. That is, sector I corruption occurs in the transfer **142***a* of the contents of a segment **146***b* to a sector **150**. The distinction of sector **150** and segment **146***b* is used for convenience, to distinguish a logical file **132** from a physical image or map in a storage device **140**. Nevertheless, each of the segments **146** may be expected to be of the same size as an individual sector **159**.

In addition to the sector I corruption of the last byte location **96***d* in the sector **150**, the conditions **136** cause sector II corruption. Thus, the contents of the segment **146***c*, when transferred **142***b* to the sector **152** of the storage device **140**, demonstrate sector II corruption as illustrated in FIG. **3**. The sector I corruption **91** affects the last byte location **96***d* of the sector **150**. The sector II corruption **93** caused by the transfer **142***a* to the sector **150** damages all of the contents of the sector **152**. As illustrated in FIG. **3**, the first byte location **98***a* of the sector **152** receives, along the path **100***e* the spurious value K **104***d*. The value K **104***d* should have been written to the last byte location **96***d* of the sector **150**. The additional characteristic of the sector II corruption **93** (long delay corruption **93**, as opposed to the short delay corruption **91**) is the placement of a value of zero as the value L **104***e* in all the remaining byte locations **98** between the second byte **94***b* and the last byte **98***d* in the sector **152**. Thus, a signature for the conditions **134** of case **1 112** and case **3 116** is the presence of the same exact value J **104***c* in the next-to-last byte location **96***c* and the last byte location **96***d* in the sector **150**. The additional signature available for case **2 114** unto the conditions **136**, is the presence of a value K **104***d* in the first byte location **98***a* of the sector **152**. The value K **104***d* is the value from the last byte location **96***d* of the segment **146***b* in the original logical file **132**. Thus, two signature features may be identified in the sectors **150, 152** indicating corruption in the transfers **142***a*, **142***b*.

In the conditions **138** or set **138**, long delay corruption **93** is present in the transfers **144***a*, **144***b* of the segments **146***b* **146***c* to respective, non-contiguous sectors **150, 154**. Accordingly, the last byte location **96***d* of the sector **150** will contain a value J **104***c* identical to that stored in the next-to-last byte location **96***c* of the sector **150**. However, since the segment **146** is written to a non-contiguous sector **154**, the long-delay corruption **93** is not present in the sector **152** subsequent to the sector **150**. Rather, a sector **154** randomly separated from the sector **150** contains the long-delay corruption **93**. Thus, case **4 118** may exist virtually anywhere in a storage device **140**.

In general, a file format managed by an operating system **64** writing to a storage device **140** controls the fragmentation of a file **132**. Periodically, defragmentation may occur. In defragmentation, the information corresponding to contiguous segments **146***b*, **146***c* may be rewritten to contiguous sectors **150, 152** in the storage device **140**. Note that the long-delay corruption **93** may occur in different ways.

For example, the contents of a segment **146***b* may be written to a sector **150** contiguously with a transfer of the segment **146***c* to the sector **152**. The long-delay corruption **93** may occur in the following sector **152**. Subsequently, the transfer **144***b* may copy the segment **146***c* to a sector **154**.

Alternatively, the sector **150** may initially receive the contents of the segment **146***b* subject to short-delay corruption **91** (in a long-delay case, short-delay corruption **91** also exists), while a designated, subsequent sector **154** receives the corrupted contents of sector II corruption **93**. The contents of the segment **146***c* may be stored as corruption in the sector **154**. Alternatively, the contents of the segment **146***c* may be stored properly in the sector **154**, with an intermediate sector **152** containing the corrupted sector II the sector **150** containing sector I corruption **91** and **93** contents.

Referring to FIG. **6**, a method **160** or process **160** is illustrated schematically for detecting corruption in a storage device **140** (see FIG. **5**). The process **160** may be thought of as a physical media scan **162**. Alternatively, one may think of a call **162** executed to run the process **160** of scanning the physical media **86** (see FIG. **3**).

Upon a call **162**, a size step **164** determines the total size of the media **86** in a storage device **140** to be tested. An initialize step **166** may set a counter to a value of zero for looping in accordance therewith.

A read step **162** may read an individual sector **88** of the media **86** in order, according to the counter **166**. Thus, the count **166** begins at zero and progresses through all sectors **88, 159**, in order. A test **170** reads the last two byte locations **96***c*, **96***d* in each sector **88, 159**. The test **170** determines whether the contents of the last byte location **96***d* are exactly equal to the contents of the next-to-last byte location **96***c*. A negative response to the test **170** indicates an inequality between the byte locations **96***c*, **96***d*. The sector I corruption **91**, or sector I corruption signature **91** is not present. Therefore, an increment step **172** increments the counter **166**. Note that a step **166** of initializing a count or creating a count loop may also be referred to as the loop or as the count itself.

If the increment **172** added to the count **166** exceeds the total number of sectors **88** in the media **86**, the test **174** will detect the end of the media **86**. A negative response to the test **174** returns the process **160** to read **168** the next, incremented sector **88** identified. A positive response to the test **174** indicates that the media **86** is completely tested, and results in a termination **176** or return **176** of the process **160**. The process **160** may operate as a standalone routine. Alternatively, the process **160** may be incorporated into other applications, such as a standard virus or corruption scanning program that searches for other types of signatures.

A positive response to the test **170** indicates that the short-delay corruption **91** appears to be present. Accordingly, a subsequent read step **178** reads the next sector **88** (e.g. sector **92**, with respect to initial sector **90**). A test **180** determines whether all of the byte locations **98**, from the second byte location **98***b* (byte **1**) through the last byte location **98***d* have a value of zero. A positive response to the test **180** indicates that long delay corruption **93** is possible. The output **182** indicates this possibility. It is also possible that the value of zero is properly written to the sector **92**. Thus, the output **182** does not necessarily indicate absolutely that long-delayed corruption **93** is present.

A negative response to the test **180** indicates that the byte locations from the second byte **98***b* to the last byte **98***d* are not all filled with a value of zero. Thus, long-delay corruption **93** does not appear to be present. Accordingly, an output **184** indicates the possibility of short-delaying corruption **91**. After the output **182, 184**, the process **160** advances by incrementing **172** the count **166** and continuing to the end of the medial **86**.

Referring to FIG. **7**, a process **190** provides a valuation of a logical file **132**. That is, the process **160** operates on a storage device **140**, and particularly on the storage medium **86** or media **86**, regardless of the nature or content of individual files **132** stored thereon. By contrast, the process **190** scans the logical files **132** in the sequence of their respective segments **146**, regardless of the nature of contiguous transfers **142** or non-contiguous transfers **144**.

The logical scan **192**, or the call **192** of a logical scan process **190**, initiates a size step **194**. The size step **194** determines the size of a particular file **132** stored in a storage device **140** (see FIG. **5**). By determining **194** the size of the file **132**, the process **190** can determine the sector-size segments **146**, with their respective boundaries **148**.

A loop **196**, or an initialize **196** may set a loop count to an initial value of zero. Such iterative processes may be implemented in a variety of ways. An initialize step **196** is one currently preferred, and simple, method. Subsequently, a read step **198** reads the segment **146** corresponding to the current count **196**. As discussed previously, the segments **146** each correspond to a sector. Nevertheless, in order to distinguish a logical sector **146** in a file **132**, the sectors **146** are referred to as segments **146**. Thus, a read **198** reads the logical sector **146** (segment **146**) corresponding to the current count **196**.

Thereafter, a test **200** determines whether the values stored in the byte locations **98** from the second byte location **98***b* to the last byte location **98***d* are all zero. A positive response to the test **200** indicates that long-delay corruption **93** is possible. Accordingly, an output **202** provides this feedback from the test **200**.

A negative response to the test **200** indicates that the contents of the byte locations **98***b* through **98***d* do not all have a value of zero. Accordingly, a test **204** follows the test **200**. The test **204** determines whether the last byte location **96***d* in a sector **88** of interest, has a value identical to that of a next-to-last byte location **96***c*. A positive response to the test **204** indicates that short-delay corruption **91** is possible. Thus, an output **206** is provided in response to the test **204**. The output **206** indicates the possibility of short-delay corruption **91**, whether or not the long-delay corruption **93** might also be present according to the output **202**.

Regardless of the outputs **202**, **206**, a subsequent increment step **208** increments the count **196** or loop **196** to advance the tests **200**, **204** to the next sector number available. If the number of the next count **196** is greater than the total size **194** determined by the size step **194**, then the test **210** so detects. That is, the test **210** determines whether the end of the file **132** has been read **198**. A negative response to the test **210** returns the process **190** to read **198** the next available sector **146** in the file **132**. Note that in each case, a segment **146** or sector **146** in the logical file **132** will still correspond to some particular sector **159** on a storage device **140**. Thus, a sector **88** of some physical medium **86** must always be read for the contents of any individual segment **146** (sector **146** logically). However, tapes, hard drives, volatile or other random access memory **20** may also be tested, and need not be arranged by the sector scheme or other physical media **86**.

A positive response to the test **210** results in a return **212** or a completion **212** of the process **190**. Accordingly, the outputs **202**, **206** may be provided in written, numerical, automated statistical, or other formats. Alternatively, the return **212** may result in automatic correction of the corruption **91**, **93** in certain instances.

Wherefore, I claim:

1. An apparatus for detecting data corruption resulting from defective operation of a floppy diskette controller, the method comprising:

a storage medium containing data disposed in a series of bytes;

a processor operably connected to the storage medium and programmed to execute a signature detection module, the signature detection module being effective to detect improper storage of the bytes, wherein the improper storage results from an error of a type causing erroneous replication of a byte in a sector of a storage medium;

a memory device operably connected to the processor for storing the signature detecting module.

2. The apparatus of claim **1**, wherein the memory device stores a filter module for filtering out false positive results, and wherein a false positive result erroneously reflects a replication of bytes properly stored.

3. The apparatus of claim **1**, wherein the storage medium contains data reflecting erroneous replication of bytes undetected at a time corresponding to a first writing.

4. The apparatus of claim **3**, wherein the erroneous replication remains undetected during at least one subsequent writing after the time of the first writing.

5. The apparatus of claim **4**, wherein the at least one subsequent writing reflects a change in type of the storage medium.

6. The apparatus of claim **5**, wherein the time of first writing is comparatively distant from a time corresponding to the subsequent writing.

7. The apparatus of claim **6**, wherein the comparative distance between the first writing and the subsequent writing reflects a loss of identifying data for detecting the error, the loss occurring after the time of first writing.

8. The apparatus of claim **4**, wherein the time of first writing occurs during a write operation to a magnetic medium.

9. The apparatus of claim **8**, wherein the magnetic medium is a floppy diskette.

10. The apparatus of claim **1**, wherein the storage medium is configured in a computer-readable data storage device.

11. The apparatus of claim **1**, wherein the storage medium is selected from the group consisting of magnetic media, optical media, magneto-optical media, and electronic media.

12. A method for detecting data corruption resulting from defective operation of a floppy diskette controller, the method comprising:

writing data as data bytes to a storage medium;

identifying a demarcation rule effective to reflect a correspondence of data bytes to the sector size of the storage medium and effective to identify a first byte, a last byte, and a next-to-last byte corresponding to the sector size;

scanning the data bytes in accordance with the demarcation rule; and

detecting erroneous replication of bytes within a segment of data corresponding to the sector size.

13. The method of claim **12**, wherein detecting further comprises comparing a value of the last byte, with a value of the next-to-last byte.

14. The method of claim **12**, further comprising applying a false positive filter to reduce spurious results.

15. The method of claim **14**, further comprising comparing previous identical values in the segment with the value of the last and next-to-last bytes.

16. The method of claim **14**, wherein the value of the last and next-to-last bytes is zero.

17. The method of claim **12**, wherein the demarcation rule reflects a number of bytes corresponding to a sector of an initial storage medium controlled by the floppy diskette controller.

18. The method of claim **17**, wherein the initial storage medium is a magnetic storage medium.

19. The method of claim **18**, wherein the magnetic storage medium is a floppy diskette.

20. The method of claim **12**, wherein the data in the storage medium is formatted in accordance with a format scheme distinct from a sectoring scheme applied at the time first writing.

21. The method of claim **20**, further comprising comparing values corresponding to a last byte and a next-to-last byte in a segment corresponding to the sector size in the sectoring scheme.

22. The method of claim **12**, wherein scanning further comprises executing a physical scan of the storage medium,

and wherein the storage medium is an original storage medium corresponding to a first writing of the data.

23. The method of claim **22**, wherein the storage medium is attached to the floppy diskette controller.

24. The method of claim **23**, wherein the magnetic storage medium is a floppy diskette.

25. The method of claim **22** further comprising filtering out false positive results.

26. The method of claim **23**, wherein the storage medium is a magnetic storage medium.

* * * * *

Exhibit D

US006401222B1

(12) **United States Patent**
Adams

(10) **Patent No.: US 6,401,222 B1**
(45) **Date of Patent: *Jun. 4, 2002**

(54) DEFECTIVE FLOPPY DISKETTE CONTROLLER DETECTION APPARATUS AND METHOD

(76) Inventor: **Phillip M. Adams**, 313 Pleasant Summit Dr., Henderson, NV (US) 89012

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **09/211,574**

(22) Filed: **Dec. 14, 1998**

Related U.S. Application Data

(63) Continuation-in-part of application No. 08/729,172, filed on Oct. 11, 1996, now Pat. No. 5,983,002.

(51) **Int. Cl.**^{7} **H02H 3/05**
(52) **U.S. Cl.** **714/42**; 713/200; 713/2
(58) **Field of Search** 714/42, 40, 25, 714/11, 8; 713/2, 200

(56) **References Cited**

U.S. PATENT DOCUMENTS

3,908,099 A		9/1975	Borbas et al.	179/175.2
4,789,985 A		12/1988	Akahoshi et al.	371/11
4,942,606 A		7/1990	Kaiser et al.	380/4
4,996,690 A		2/1991	George et al.	371/37.1
5,093,910 A		3/1992	Tulpule et al.	395/575
5,212,795 A		5/1993	Hardry	395/725
5,233,692 A		8/1993	Gajjar et al.	395/325
5,237,567 A		8/1993	Nay et al.	370/85.1
5,379,414 A		1/1995	Adams	395/575
5,416,782 A		5/1995	Wells et al.	371/21.2
5,422,892 A		6/1995	Hii et al.	371/24
5,442,753 A		8/1995	Waldrop et al.	395/842
5,544,334 A	*	8/1996	Noll	395/309
5,619,642 A		4/1997	Nielson et al.	395/182.04
5,649,212 A		7/1997	Kawamura et al.	395/570
5,666,540 A		9/1997	Hagiwara et al.	395/750.05

OTHER PUBLICATIONS

NEC Electronics, Inc.,, "IBM–NEC Meeting for μPD765A/μPD72065 Problem" (U.S.A., May 1987).

Intel Corporation, Letter to customer from Jim Sleezer, Product Manager, regarding FDC error and possible solutions (U.S.A., May 2, 1988).

Adams, Phillip M., Nova University, Department of computer Science, "Hardware–Induced Data Virus," Technical Report TR–881101–1 (U.S.A. Nov, 14, 1988).

Advanced Military Computing, "Hardware Virus Threatens Database," vol. 4, No. 25, pp. 1&8 (U.S.A. Dec. 5, 1988).

Intel Corporation, "8237A/8237A–4/8237A–5 High Performance Programmable DMA Controller" (U.S.A., date unknown).

Intel Corporation, "8272A Single/Double Density Floppy Disk Controller" (U.S.A., date unknown).

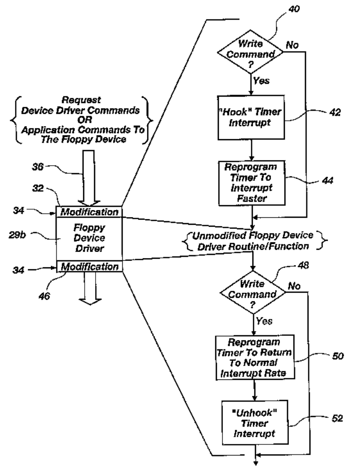
* cited by examiner

Primary Examiner—Dieu-Minh Le
Assistant Examiner—Rita A Ziemer
(74) *Attorney, Agent, or Firm*—Pate Pierce & Baird

(57) **ABSTRACT**

A system and method which provides a complete software implementation of a detection process that is capable of detecting defective Floppy Diskette Controllers ("FDCs") without visual hardware inspection or identification. The approach taken includes a multi-phase strategy incorporating programmatic FDC identification, software DMA shadowing, defect inducement, and use of a software decoding network which allows the implementation of the invention to adjust to a wide range of computer system performance levels. A method and apparatus for detecting and preventing floppy diskette controller data transfer errors in computer systems is also provided. The approach taken may involve software DMA shadowing and the use of a software decoding network.

20 Claims, 7 Drawing Sheets


Request Device Driver Commands OR Application Commands To The Floppy Device
36
32
34
Modification
29b
Floppy Device Driver
34
Modification
46
40
Write Command ?
No
Yes
"Hook" Timer Interrupt
42
Reprogram Timer To Interrupt Faster
44
Unmodified Floppy Device Driver Routine/Function
48
Write Command ?
No
Yes
Reprogram Timer To Return To Normal Interrupt Rate
50
"Unhook" Timer Interrupt
52

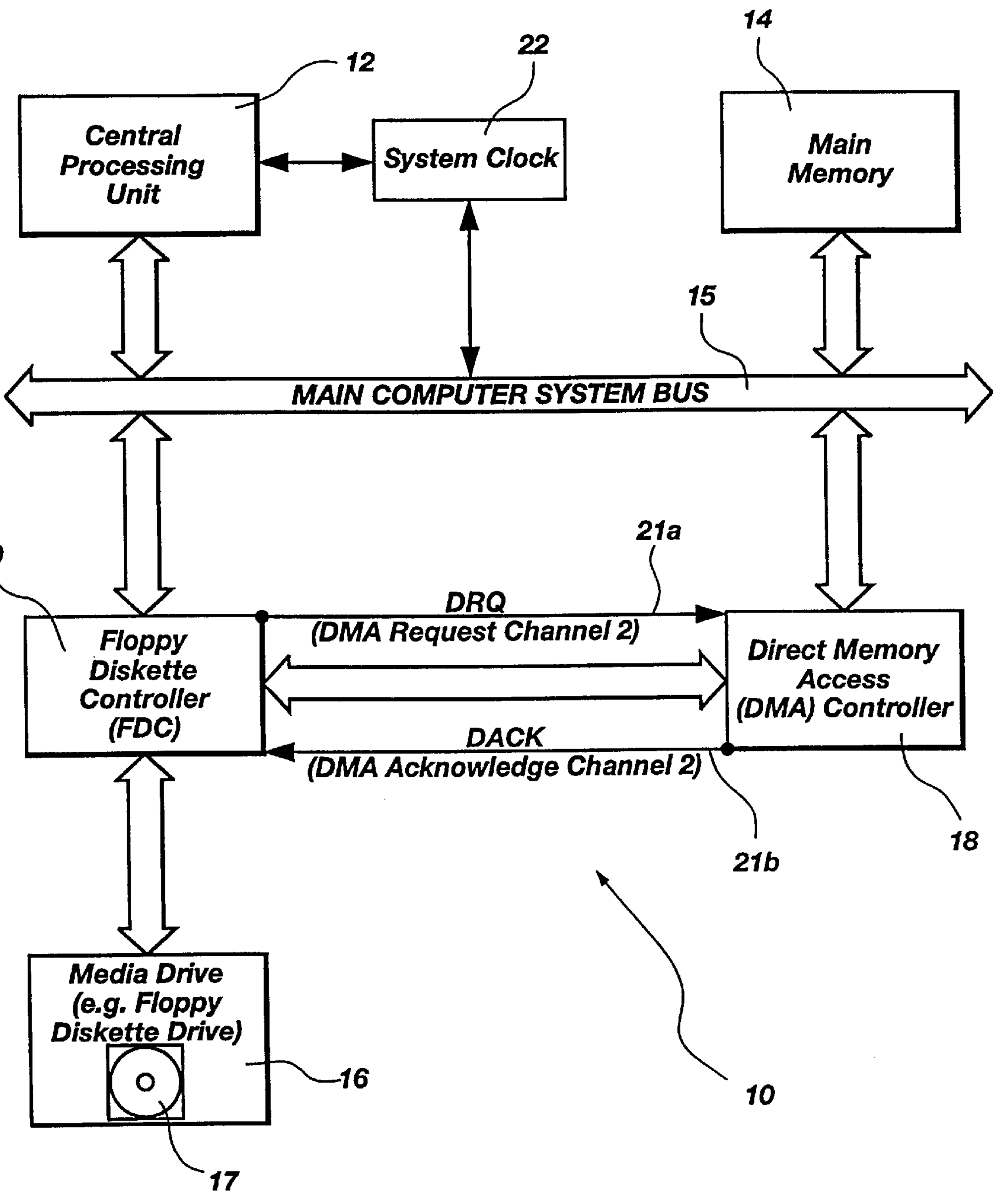

12
22
14
Central Processing Unit
System Clock
Main Memory
15
MAIN COMPUTER SYSTEM BUS
21a
20
DRQ
(DMA Request Channel 2)
Floppy Diskette Controller (FDC)
Direct Memory Access (DMA) Controller
DACK
(DMA Acknowledge Channel 2)
21b
18
Media Drive (e.g. Floppy Diskette Drive)
16
17
10

Fig. 1

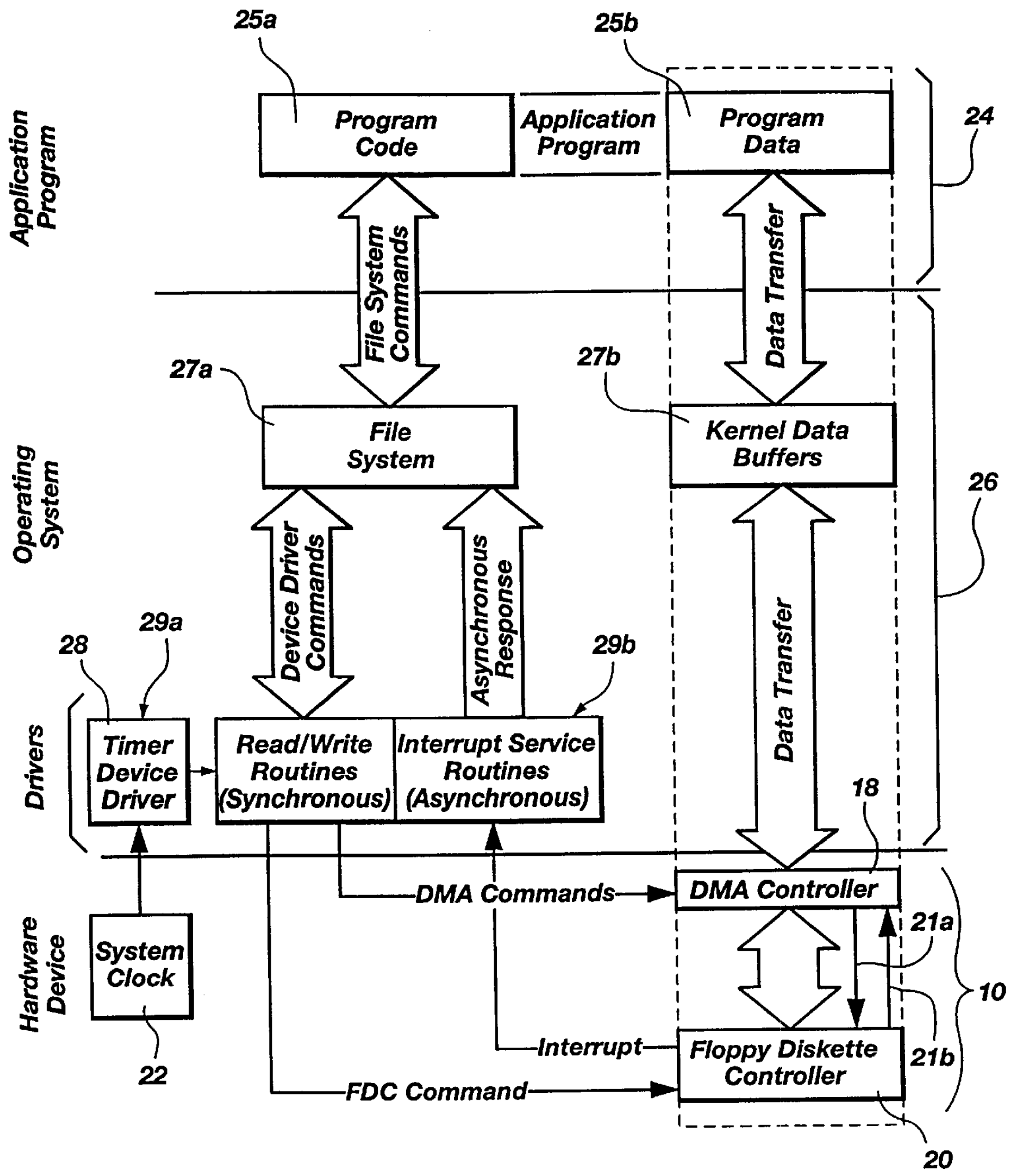

25a
25b
Application Program
Program Code
Application Program
Program Data
24
File System Commands
Data Transfer
27a
27b
Operating System
File System
Kernel Data Buffers
26
Device Driver Commands
Asynchronous Response
Data Transfer
28
29a
29b
Drivers
Timer Device Driver
Read/Write Routines (Synchronous)
Interrupt Service Routines (Asynchronous)
18
DMA Commands
DMA Controller
21a
Hardware Device
System Clock
10
Interrupt
Floppy Diskette Controller
21b
22
FDC Command
20

Fig. 2

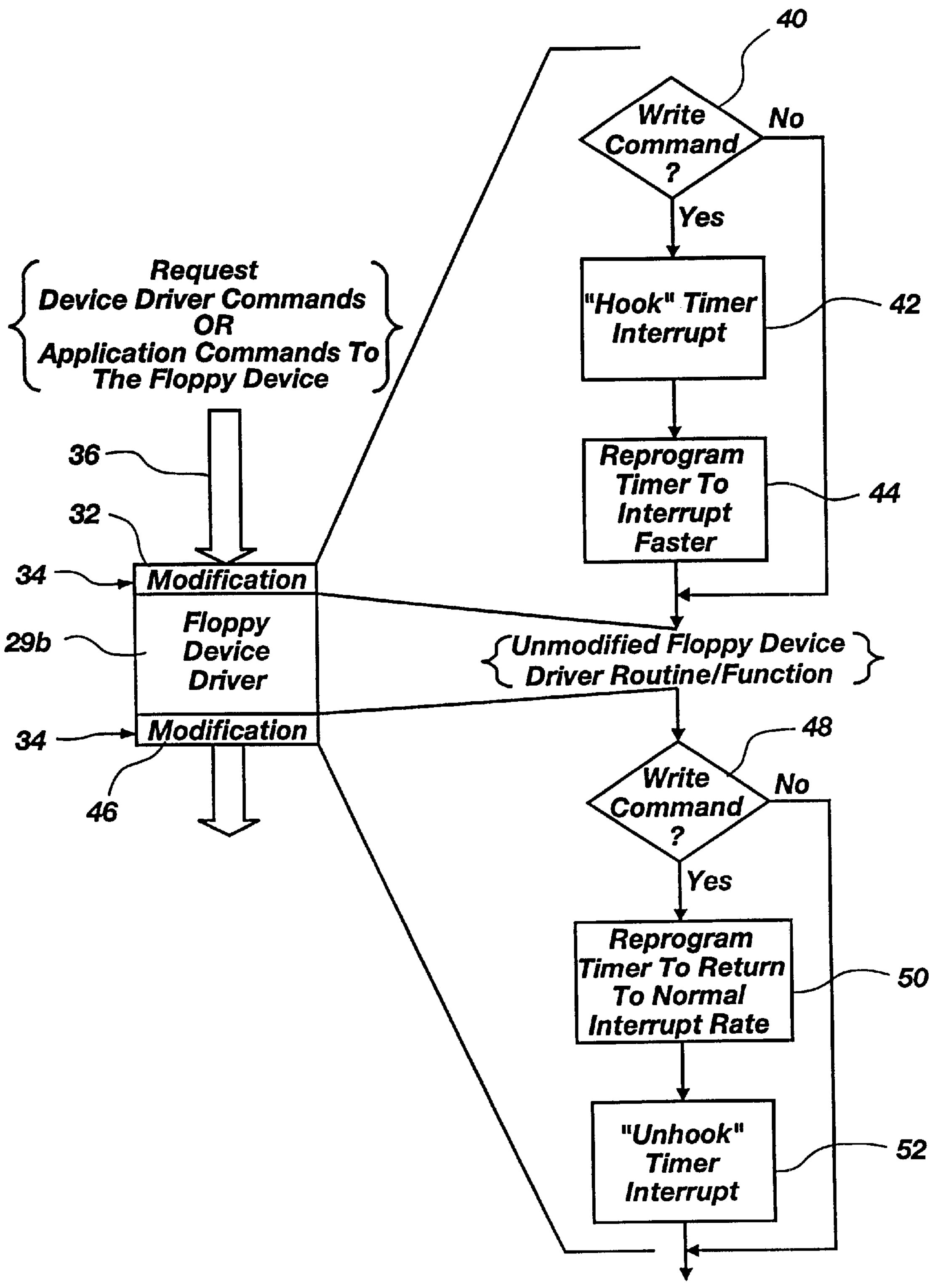

Request Device Driver Commands OR Application Commands To The Floppy Device
36
32
34
Modification
29b
Floppy Device Driver
34
Modification
46
40
Write Command ?
No
Yes
"Hook" Timer Interrupt
42
Reprogram Timer To Interrupt Faster
44
Unmodified Floppy Device Driver Routine/Function
48
Write Command ?
No
Yes
Reprogram Timer To Return To Normal Interrupt Rate
50
"Unhook" Timer Interrupt
52

Fig. 3

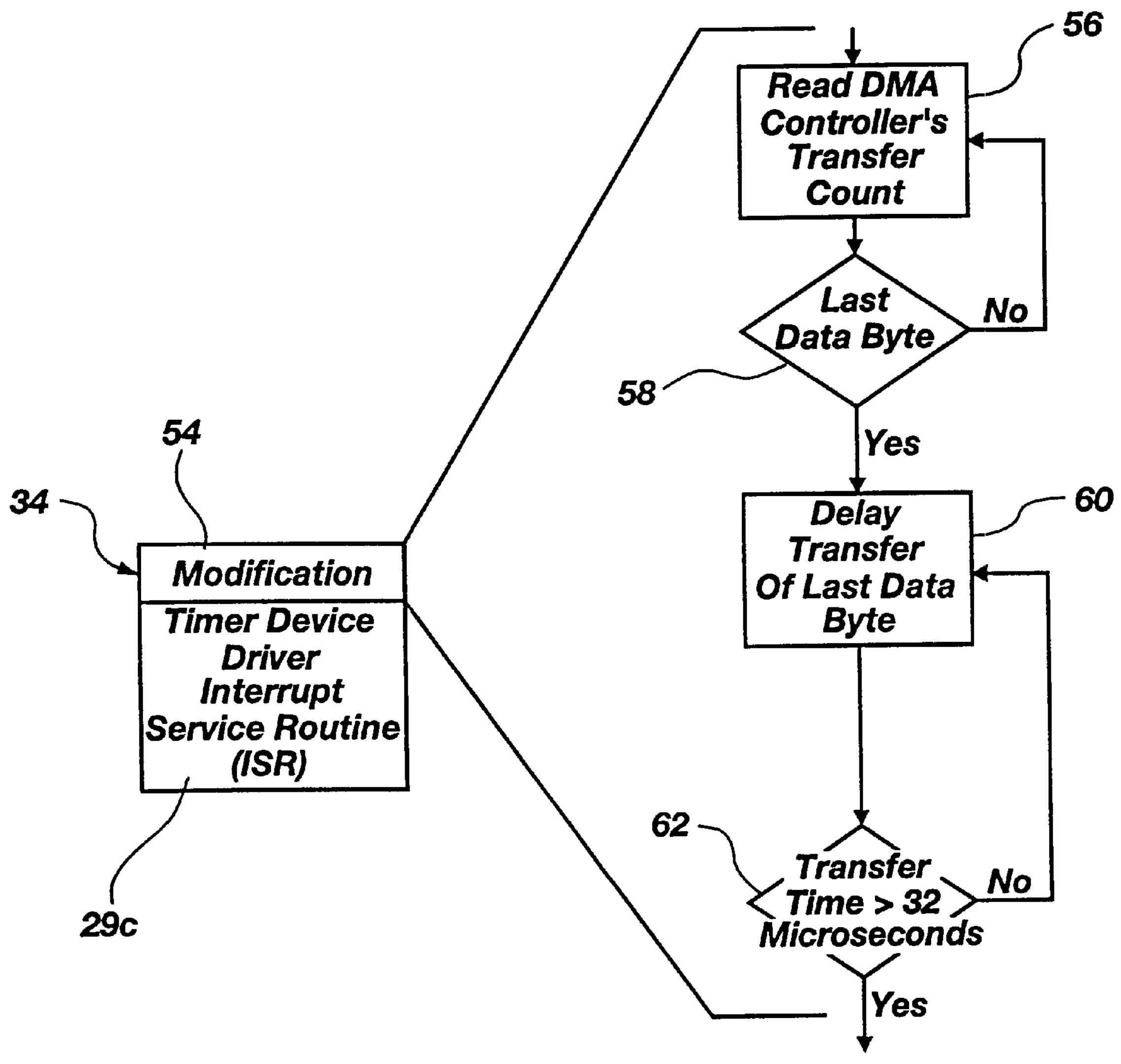

56
Read DMA Controller's Transfer Count
Last Data Byte
No
58
Yes
60
Delay Transfer Of Last Data Byte
54
34
Modification
Timer Device Driver Interrupt Service Routine (ISR)
29c
62
Transfer Time > 32 Microseconds
No
Yes

Fig. 4

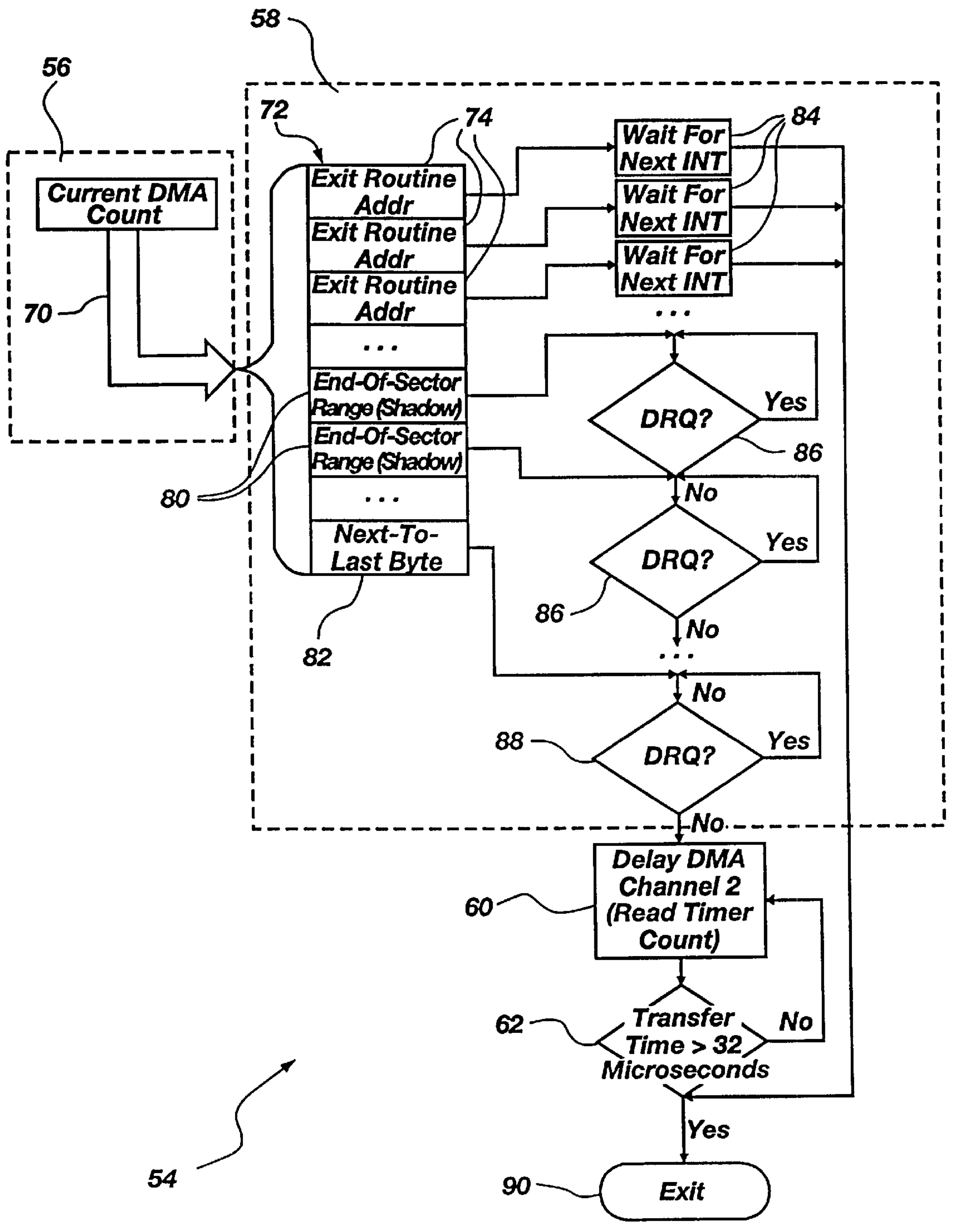

58
56
72
74
84
Current DMA Count
Exit Routine Addr
Exit Routine Addr
Exit Routine Addr
Wait For Next INT
Wait For Next INT
Wait For Next INT
70
End-Of-Sector Range (Shadow)
End-Of-Sector Range (Shadow)
DRQ?
Yes
No
86
80
Next-To-Last Byte
82
88
Delay DMA Channel 2 (Read Timer Count)
60
62
Transfer Time > 32 Microseconds
No
Yes
54
90
Exit

Fig. 5

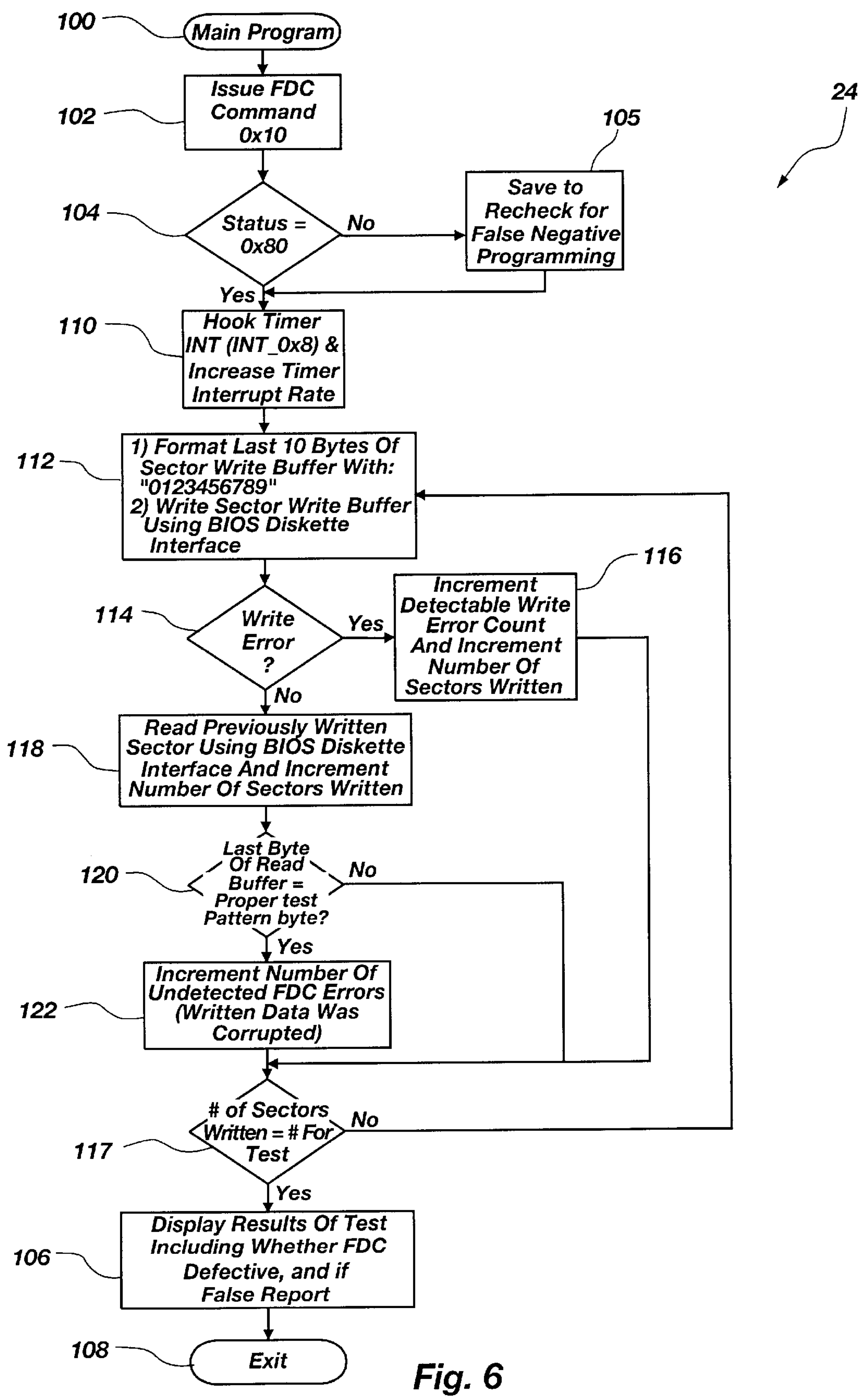

100
Main Program
102
Issue FDC Command 0x10
104
Status = 0x80
No
105
Save to Recheck for False Negative Programming
Yes
24
110
Hook Timer INT (INT_0x8) & Increase Timer Interrupt Rate
112
1) Format Last 10 Bytes Of Sector Write Buffer With: "0123456789"
2) Write Sector Write Buffer Using BIOS Diskette Interface
114
Write Error ?
Yes
116
Increment Detectable Write Error Count And Increment Number Of Sectors Written
No
118
Read Previously Written Sector Using BIOS Diskette Interface And Increment Number Of Sectors Written
120
Last Byte Of Read Buffer = Proper test Pattern byte?
No
Yes
122
Increment Number Of Undetected FDC Errors (Written Data Was Corrupted)
117
# of Sectors Written = # For Test
No
Yes
106
Display Results Of Test Including Whether FDC Defective, and if False Report
108
Exit

Fig. 6

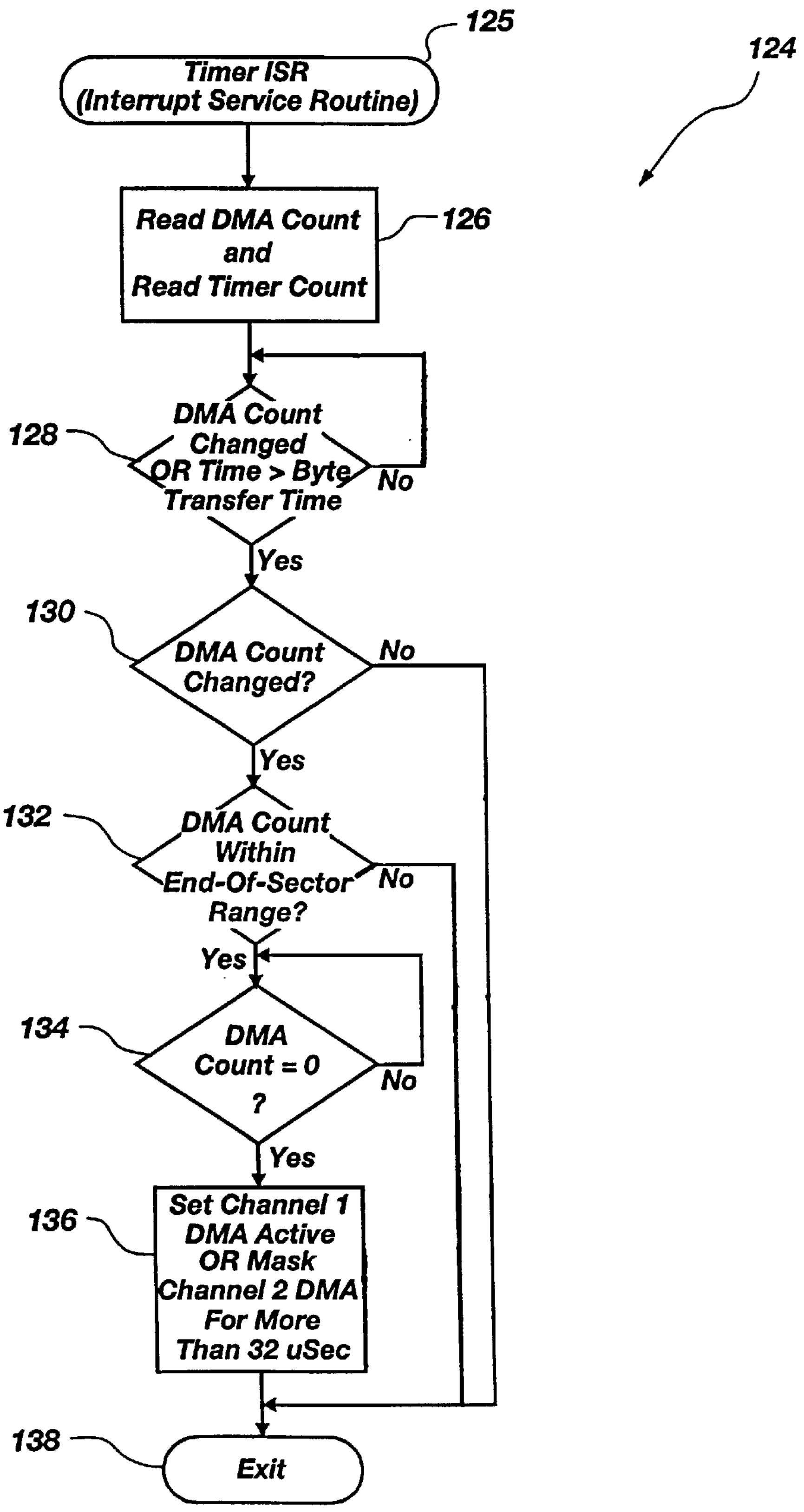

125
124
Timer ISR
(Interrupt Service Routine)
Read DMA Count and Read Timer Count
126
128
DMA Count Changed OR Time > Byte Transfer Time
No
Yes
130
DMA Count Changed?
No
Yes
132
DMA Count Within End-Of-Sector Range?
No
Yes
134
DMA Count = 0 ?
No
Yes
136
Set Channel 1 DMA Active OR Mask Channel 2 DMA For More Than 32 uSec
138
Exit

Fig. 7

DEFECTIVE FLOPPY DISKETTE CONTROLLER DETECTION APPARATUS AND METHOD

RELATED INVENTIONS

This application is a Continuation in Part of my co-pending application Ser. No. 08/729,172 filed on Oct. 11, 1996 now U.S. Pat. No. 5,983,002 for Defective Floppy Diskette Controller Detection Apparatus and Method.

BACKGROUND

1. The Field of the Invention

This invention relates to the detection of defective Floppy Diskette Controllers ("FDCs") where an undetected data error causes data corruption and, more particularly, to novel systems and methods implemented as a software-only detection mechanism which eliminates the need for visual inspection or identification of the FDCs.

2. The Background Art

Computers are now used to perform functions and maintain data critical to many organizations. Businesses use computers to maintain essential financial and other business data. Computers are also used by government to monitor, regulate, and even activate, national defense systems. Maintaining the integrity of the stored data is essential to the proper functioning of these computer systems, and data corruption can have serious (even life threatening) consequences.

Most computer systems include media drives, such as floppy diskette drives for storing and retrieving data. For example, an employee of a large financial institution may have a personal computer that is attached to the main system. In order to avoid processing delays on the mainframe, the employee may routinely transfer data files from a host system to a local personal computer and then back again, temporarily storing or backing up data on a local floppy diskette or other media. Similarly, an employee with a personal computer at home may occasionally decide to take work home, transporting data away from and back to the office on a floppy diskette.

Data transfer to and from media, such as a floppy diskette, is controlled by a device called a Floppy Diskette Controller ("FDC"). The FDC is responsible for interfacing the computer's Central Processing Unit ("CPU") with a physical media drive. Significantly, since the drive is spinning, it is necessary for the FDC to provide data to the drive at a specified data rate. Otherwise, the data will be written to a wrong location on the media.

The design of an FDC accounts for situations occurring when a data rate is not adequate to support rotating media. Whenever this situation occurs, the FDC aborts the write operation and signals to the CPU that a data under run condition has occurred.

Unfortunately, however, it has been found that a design flaw in many FDCs makes impossible the detection of certain data under run conditions. This flaw has, for example, been found in the NECK 765, INTEL 8272 and compatible Floppy Diskette Controllers. Specifically, data loss and/or data corruption may routinely occur during data transfers to or from diskettes (or tape drives and other media attached via the FDC), whenever the last data byte of a sector being transferred is delayed for more than a few microseconds. Furthermore, if the last byte of a sector write operation is delayed too long then the next (physically adjacent) sector of the media will be destroyed as well.

For example, it has been found that these faulty FDCs cannot detect a data under run on the last byte of a diskette read or write operation. Consequently, if the FDC is preempted or otherwise suspended during a data transfer to the diskette (thereby delaying the transfer), and an under run occurs on the last byte of a sector, the following occur: (1) the under run flag does not get set, (2) the last byte written to the diskette is made equal to either the previous byte written or zero, and (3) a successful Cyclic Redundancy Check ("CRC") is generated on the improperly altered data. The result is that incorrect data is written to the diskette and validated by the FDC. Herein, references to a floppy diskette may be read as "any media" and a floppy diskette drive is but a specific example of a media drive controllable by an FDC.

Conditions under which this problem may occur have been identified in connection with the instant invention by identifying conditions that can delay data transfer to or from the diskette drive. In general, this requires that the computer system be engaged in "multi-tasking" operation or in overlapped input/output ("I/O") operation. Multi-tasking is the ability of a computer operating system to simulate the concurrent execution of multiple tasks.

Importantly, concurrent execution is only "simulated" because only one CPU exists in a typical personal computer. One CPU can only process one task at a time. Therefore, a system interrupt is used to rapidly switch between the multiple tasks, giving the overall appearance of concurrent execution.

MS-DOS and PC-DOS, for example, are single-task operating systems. Therefore, one could argue that the problem described above would not occur. However, a number of standard MS-DOS and PC-DOS operating environments simulate multi-tasking and are susceptible to the problem.

In connection with the instant invention, for example, the following environments have been found to be prime candidates for data loss and/or data corruption due to defective FDCs: local area networks, **327**x host connections, high density diskettes, control print screen operations, terminate and stay resident ("TSR") programs. The problem also occurs as a result of virtually any interrupt service routine. Thus, unless MS-DOS and PC-DOS operating systems disable all interrupts during diskette transfers, they are also highly susceptible to data loss and/or corruption.

The UNIX operating system is a multi-tasking operating system. It has been found, in connection with the instant invention, how to create a situation that can cause the problem within UNIX. One example is to begin a large transfer to the diskette and place that transfer in the background by beginning to process the contents of a very large file in a way that requires the use of a Direct Memory Access ("DMA") channel of a higher priority than that of the floppy controller's DMA channel. These higher priority processes might include, for example, video updates, multi-media activity, etc. Video access forces the video buffer memory refresh logic on DMA channel **1**, along with the video memory access, which preempts the FDC operations from occurring on DMA channel **2** (which is lower priority than DMA channel **1**).

This type of example creates an overlapped I/O environment and can force the FDC into an undetectable error condition. More rigorous examples include a concurrent transfer of data to or from a network or tape drive using a high priority DMA channel while the diskette transfer is active. Clearly, the number of possible error producing examples is infinite, yet each is highly probable in this environment.

For all practical purposes the OS/2 and newer Windows operating systems can be regarded as UNIX derivatives. They suffer from the same problems that UNIX does. Two significant differences exist between these operating systems and UNIX.

First, they both semaphore video updates with diskette operations tending to avoid forcing the FDC problem to occur. However, any direct access to the video buffer, in either real or protected mode, during a diskette transfer will bypass this feature and result in the same faulty condition as UNIX.

Second, OS/2 incorporates a unique command that tends to avoid the FDC problem by reading back every sector that is written to the floppy diskette in order to verify that the operation completed successfully. This command is an extension to the MODE command (MODE DSKT VER= ON). With these changes, data loss and/or data corruption should occur less frequently than otherwise. However, the FDC problem may still destroy data that is not related to the current sector operation.

A host of other operating systems are susceptible to the FDC problem just as DOS, Windows, Windows 95, Windows 98, Windows NT, OS/2, and UNIX. However, these systems may not have an installed base as large as DOS, Windows, OS/2 or UNIX, and may, therefore, receive less motivation to address the problem. Significantly, as long as the operating systems utilize the FDC and service system interrupts, the problem can manifest itself This can occur in computer systems that use virtually any operating system.

Some in the computer industry have suggested that data corruption by the FDC is extremely rare and difficult to reproduce. This is similar to the argument presented during the highly publicized 1994 defective INTEL Pentium scenario. Error rate frequencies for the defective Pentium ranged from microseconds to tens-of-thousands of years! The FDC problem is often very difficult to detect during normal operation because of its random characteristics. The only way to visibly detect this problem is to have the FDC corrupt data that is critical to the operation at hand. However, many locations on the diskette may be corrupted, yet not accessed. In connection with the instant invention, the FDC problem has been routinely reproduced and may be more common than heretofore believed.

Computer users may, in fact, experience this problem frequently and not even know about it. After formatting a diskette, for example, the system may inform the user that the diskette is bad, although the user finds that if the operation is performed again on the same diskette everything is fine. Similarly, a copied file may be unusable, and the computer user concludes that he or she just did something wrong. For many in this high-tech world, it is very difficult to believe that the machine is in error and not themselves. It remains typical, however, that full diskette back-ups are seldom restored, that all instructions in programs are seldom, if ever, executed, that diskette files seldom utilize all of the allocated space, and that less complex systems are less likely to exhibit the problem.

Additionally, the first of these faulty FDCs was shipped in the late 1970's. The devices were primarily used at that time in special-purpose operations in which the FDC problem would not normally be manifest. Today, on the other hand, the FDCs are incorporated into general-purpose computer systems that are capable of concurrent operation (multitasking or overlapped I/O). Thus, it is within today's environments that the problem is most likely to occur by having another operation delay a data transfer to a diskette. The

more complex a computer system, the more likely it is that one activity will delay another, thereby creating an FDC error condition.

In short, the potential for data loss and/or data corruption is present in all computer systems that utilize the defective version of this type of FDC, presently estimated at about 50 million personal computers. The design flaw in the FDC causes data corruption to occur and manifest itself in the same manner as a destructive computer virus. Furthermore, because of its nature, this problem has the potential of rendering even secure databases absolutely useless.

Various conventional ways of addressing the FDC problem, such as a hardware recall, have significant associated costs, risks and/or disadvantages. In addition to a solution to the FDC problem, an apparatus and method are needed to accurately, rapidly, reliably, and correctly, identify any defective FDC. The identification of defective FDCs is the first step in attempting to solve the problem of defective FDCs. A solution method and apparatus for repairing a defective FDC are disclosed in U.S. Pat. No. 5,379,414 incorporated herein by reference.

BRIEF SUMMARY AND OBJECTS OF THE INVENTION

In view of the foregoing, it is a primary object of the present invention to provide a method and apparatus for detecting defective Floppy Diskette Controllers ("FDCs").

It is another object of the present invention to provide a software (programmatic) solution that may be implemented in a general purpose digital computer, which eliminates the need for visual inspection and identification of the defective FDCs as well as the need for any hardware recall and replacement.

Consistent with the foregoing objects, and in accordance with the invention as embodied and broadly described herein, an apparatus and method are disclosed in one embodiment of the present invention as including data structures, executable modules, and hardware, implementing a detection method capable of immediately, repeatedly, correctly, and accurately detecting defective FDCs.

The apparatus and method may rely on 1) determining whether or not the FDC under test is a new model FDC (potentially non-defective), and 2) if the FDC under test is not a new model FDC, installing an interposer routine to force the FDC to delay a transfer of a last data byte of a sector either to or from the floppy diskette whose controller is tested. A test condition is thus created in the hardware to cause defective FDCs to corrupt the last data byte of the sector. A second portion of an apparatus and method may confirm a diagnosis. Thus the apparatus and method may ensure that old-model non-defective FDCs are not wrongly identified as defective.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects and features of the present invention will become more fully apparent from the following description and appended claims, taken in conjunction with the accompanying drawings. Understanding that these drawings depict only typical embodiments of the invention and are, therefore, not to be considered limiting of its scope, the invention will be described with additional specificity and detail through use of the accompanying drawings in which:

FIG. **1** is a schematic block diagram of an apparatus illustrating the architecture of a computer system for testing a floppy diskette controller ("FDC")in accordance with the invention;

FIG. **2** is a schematic block diagram illustrating software modules executing on the processor and stored in the memory device of FIG. **1**, including application programs, operating systems, device drivers and computer system hardware such as a floppy diskette;

FIG. **3** is a schematic block diagram of a flow chart depicting one presently preferred embodiment of certain modifications that may be applied to a diskette device driver in order to force an otherwise undetected error condition to occur in a defective FDC, thus enabling the defective FDC detection apparatus and method of the present invention to be activated,

FIG. **4** is a schematic block diagram of a flow chart depicting one presently preferred embodiment of certain modifications that may be made to a timer Interrupt Service Routine ("ISR") to allow timing of a transfer byte's DMA request and DMA acknowledge (DREQ/DACK) cycle in order to ensure that proper conditions exist to create data corruption associated with defective FDCs in accordance with the present invention;

FIG. **5** is a schematic block diagram of a flow chart depicting one presently preferred embodiment of a software decoding network (software vector-table) for use in connection with a defective FDC detection apparatus and method in accordance with the present invention, the software decoding network having one code point/entry for each possible transfer byte in a sector;

FIG. **6** is a schematic block diagram of a flow chart depicting one presently preferred embodiment of an application implementation of the apparatus and method of FIGS. **3** and **4**, wherein a main "driver" portion of an application forces an undetected error condition in a defective FDC enabling activation of a the defective FDC detection system in accordance with the invention; and

FIG. **7** is a schematic block diagram of a flow chart depicting one presently preferred embodiment of certain modifications that may be made to a timer Interrupt Service Routine embedded within the application of FIG. **6** to allow timing of a last byte's DREQ/DACK cycle, ensuring that proper conditions exist to create data corruption associated with defective FDCs in accordance with the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

It will be readily understood that the components of the present invention, as generally described and illustrated in the Figures herein, could be arranged and designed in a wide variety of different configurations. Thus, the following more detailed description of the embodiments of the system apparatus and method of the present invention, as represented in FIGS. **1** through **7**, is not intended to limit the scope of the invention, as claimed, but it is merely representative of the presently preferred embodiments of the invention.

The presently preferred embodiments of the invention will be best understood by reference to the drawings, wherein like parts are designated by like numerals throughout.

The architecture of an apparatus **10**, including a computer system implementing one embodiment of the invention is illustrated in FIG. **1**. A Central Processing Unit ("CPU") **12** and main memory **14** may be connected by a bus **15** inside a computer system unit. Instructions (executables) and data structures used by the CPU **12** are kept in main memory **14** during computer work sessions. Main memory **14** is,

however, not a permanent storage place for information; it is active only when the apparatus **10** (computer system) is powered up (on). Thus, to avoid losing data, data must be saved on some type of non-volatile storage device. For example, the apparatus may use a "hard disk" storage device permanently installed in the computer system. A computer system **10** may have at least one floppy diskette drive **16** that receives a removable floppy diskette (magnetic storage medium). The floppy diskette likewise may be used for "permanent" (non-volatile) storage of data or software (executables) outside of the computer system **10**. Flexible (floppy) diskettes are especially useful for transferring data and information between separate computer systems **10**.

In transferring data to a floppy diskette, the CPU **12** may program a Direct Memory Access ("DMA") controller **18** for an input/output ("I/O") transfer. The CPU **12** issues a command to a Floppy Diskette Controller ("FDC") **20** to begin the I/O transfer, and then waits for the FDC **20** to interrupt the CPU **12** with a completion interrupt signal. It is also possible to perform Programmed I/O ("PIO") directly between the CPU **10** and the FDC **20** without involving the DMA controller **18**. This latter approach is seldom used; the majority of computer systems **10** employ DMA for I/O transfers to and from the floppy diskette drive **16**. The invention will thus be described below with particular reference to the DMA controller **18**. If PIO is employed, however, then an I/O transfer is totally controlled by the CPU **12** because the CPU **12** is required to pass each and every data byte to the FDC **16**. As a result, the "DMA shadowing" system and method in accordance with the invention may be directly applied to a PIO data stream. This is readily tractable because the CPU **12** already is controlling the I/O transfer, as will become more readily apparent.

A computer system **10** may have a system clock **22**. The system clock **22** is beneficial when initiating an I/O transfer to the diskette drive **16** because one must not only control the data transfer, but also a drive motor. In this regard, it is important to know when the diskette drive motor has brought a diskette's spin rate up to a nominal RPM required for a data transfer to be successful.

For example, in IBM Personal Computers and "compatibles," the system clock **22** interrupts the CPU **12** at a rate of 18.2 times per second (roughly once every 54.9 milliseconds). This interrupt is used to determine such things as diskette drive motor start and stop time. There are also a host of other time-dependent operations in the computer system **10** that require this granularity of timing.

One presently preferred embodiment of an association between application programs **24** (executables), operating systems **26**, device drivers and hardware is depicted in FIG. **2**. The example presented corresponds to a floppy diskette having a controller **16**.

A system suitable for implementing the invention may include an application program **24** including both executable code **25*a*** and associated data **25*b***. The application **24** may interface with the hardware apparatus **10** through an operating system **26**. The operating system may include a file system **27*a*** as well as selected buffers **27*b***. The file system **27*a*** may include an executable for file system management as well as operating system interfacing.

The file system **27*a*** may issue commands to drivers **28**.

The drivers **28** may include a timer device driver **29*a***, including a timer ISR, interfacing to the system clock **22**. Likewise, a media drive driver **29*b***, alternatively referred to as a media driver **29*b*** may be included. The media driver may interface with a floppy diskette drive or other media drive **16** to maintain persistent storage on media **17**. Although a media drive **16** may typically relate to floppy diskettes, tape drives and other magnetic media may also be used in an apparatus and method in accordance with the invention.

The media driver **29***b* may be responsible for sending instructions and control signals to the media drive controller **20**, which is typically embodied as a floppy diskette controller **20**. Similarly, the media driver **29***b* may instruct and control the DMA controller **18**. The DMA controller manages data transfers between the floppy diskette controller (FDC **20**) and the main memory device **14**. A DMA request (DRQ;DREQ **21***a*) may pass from FDC controller **20** to the direct memory access controller **18** (DMA controller **18**). Likewise, a DMA acknowledge **21***b* or acknowledgment **21***b*, alternatively referred to as a (DACK **21***b*) may be returned from the DMA controller **18** to the FDC **20**.

Referring now to FIGS. **3–5**, and more FIGS. **3** and **4**, a method in accordance with the present invention include a module **32**, one of several interposer routines **34**, which is placed between an application's **24** request **36** for floppy service and a floppy device driver **29***b*. The interposer routine **32** is actually a new or modified device driver that forces certain undetected FDC data corruption conditions to exist. As shown, the interposer **32** first tests **40** whether an operation requested **36** is a floppy diskette write operation. Read operations are equally susceptible to the problem and may be used in the detection process, if desired. If so, the major function of the interposer **32** is to insert itself between the application request **36** for floppy service and the floppy device driver **29***b* that will service the request. In a PC/MS-DOS environment, this can be accomplished by "hooking" the INT 0×13 interrupt vector and directing it to the FDD prefix **32** or interposer routine **32**. Reprogramming **44** the timer **22** to interrupt faster (e.g., every 4–7 milliseconds) than normal (e.g. 54.9 milliseconds).

As will become more fully apparent from the following discussion, once a floppy write operation is detected, in a test **40** a software decoding network call vector of the timer interrupt **54** (see FIGS. **4–5**) is preferably installed. The current byte count is read **56**, and DMA shadowing **58** begins. When a test **58** shows that a current DMA transfer count (countdown) has reached 0, then the interposer routine **54** delays **60** the DMA transfer of the last byte of the sector transfer. The delay continues until a test **62** determines that the elapsed time is greater than the maximum time required for a data byte to be transferred to the medium **17** (e.g. a low-density diskette; >32 uSec).

This delay **60** forces defective FDCs **20** into an undetected data corruption condition. This condition can be tested **120** by reading back **118** the written data to see whether the last byte or the next-to-the-last byte was actually written to the last byte location of the sector.

Referring again to FIG. **3**, the system clock **22** may be reprogrammed **50** in a suffix routine **46** appended to the floppy device driver **29***b*. The system clock **22** may then interrupt normally (e.g., every 54.9 milliseconds). The timer interrupt **54** is "unhooked" **50** until the test **40** reports the next floppy write operation.

One could allow the timer **22** (clock **22**) to always interrupt at the accelerated rate. Then, a check the timer Interrupt Service Routine ("ISR")**29***c*(see FIG. **4**), within the timer device driver **29***a*, may then determine whether a media (e.g. diskette) write operation is active. Likewise, it is possible to randomly check to see if the last byte of a floppy sector write operation is in progress. However, the foregoing method has superior efficiency and accuracy in creating the condition required for the detection of defective FDCs.

As used herein, "DMA shadowing" may be thought of as programmatic CPU **12** monitoring of data (byte) transfers and timing the last byte of a sector's DREQ **21***a* to DACK **21***b* signals. Importantly, there are, of course, a number of ways of determining when the DREQ **21** a is present and when the DACK **21***b* is present. The present invention may include the use of any "DMA shadowing" whether the DREQ **21***a* and DACK **21***b* signals are detected at the DMA controller **18**, CPU **12**, system bus **15** or FDC **20**. This includes both explicit means, and implicit means.

For example, inferring the state of the DREQ/DACK cycle is possible from various components in the system that are triggered or reset from transitions of such signals **21***a*, **21***b*. In one embodiment the DACK **21***b* may cause a Terminal Count ("TC") signal to be asserted by the DMA controller **18**. Therefore, one may imply from the detection of the TC that a DACK **21***b* has occurred.

Whenever an application **24** requests a write operation of the media drive **16**, the system clock **22** may be reprogrammed to interrupt, for example, every 4 to 7 milliseconds. Referring again to FIGS. **4–5**, each time the system clock **22** interrupts, the current byte count in the transfer register (countdown register) DMA controller **18** is read **56**. Once the test **58** indicates that the byte counter has reached the last byte, the signal transition from DREQ **21***a* to DACK **21***b* may be timed and accordingly delayed **60**. This transition may be forced to be greater than the maximum time required to transfer one data byte as indicated in the test **62**.

Therefore, defective FDCs **20** are forced into an undetected data corruption state. This state may be detected by writing known data patterns to the next-to-the-last and the last data bytes. Reading the data back will reveal which of the two data bytes was stored in the last byte of the sector. Finally, it is possible to also detect defective FDCs **20** by significantly increasing the delay time during the transfer of the last byte of a sector. This forces the next physically adjacent sector to be zeroed out except for the first byte of that sector.

For the system to maintain proper operation, an interposer routine **34** should save the original INT 0×13 (Hex 13th interrupt vector) contents (address of the original INT 0×13 Interrupt Service Routine) and invoke the original when necessary. Additional aspects of the interposer function **34** are discussed below in connection with other features of the device driver **29***b*.

This implementation of the apparatus and method of the present invention is contemplated for use on an IBM Personal Computer running the PC/MS-DOS operating system. Versions have, however, been developed to operate in the Windows, OS/2 and UNIX environments and may be embodied for other operating systems. The invention is not limited to use with any particular operating system, and adaptations and changes which may be required for use with other operating systems will be readily apparent to those of ordinary skill in the art.

As depicted graphically in FIG. **4** below, a timer ISR routine **29***c* is used for servicing the accelerated interrupt rate of the system clock **22**. The reason that the system clock interrupt rate is accelerated is that during a normal 512 byte data transfer (the typical sector size) 16 microseconds are required for each data byte to be transferred to the FDC (High Density Diskette Mode). Therefore, a typical sector transfer requires 512 times 16 microseconds, or 8,192 microseconds. If the diskette is a low density diskette then the sector transfer time is doubled to 16,384 microseconds (512 times 32 microseconds) because the FDC has half of the amount of data to store in the same rotational time frame (typically 360 RPM).

Referring to FIG. **5**, the timer ISR routine **29***c* within the timer device driver **29***a* with its prefix **54** performs checks on the system **10** to determine if the system **10** is actually transferring data to the FDC **20**. If a sector transfer is not in progress then the timer ISR prefix **54** exits immediately. However, if a sector transfer is in progress then the timer ISR prefix **54** obtains the remaining byte count of the sector transfer **70** and vectors (jumps) through the software decoding network **72** (DMA count table **72**) to an appropriate processing routine **84**, **86**, **88**.

Although the steps **56**, **58** of the module **54** may be implemented with the timer **22** continually interrupting every 8, 16, or 32 microseconds. This level of interrupts may totally consume a PC's processing power, and on most PCs could not be sustained. Thus, in order to perform DMA shadowing without affecting the total system performance it is important to allow normal operations to continue as usual. It is desirable to have an interrupt (the system clock **22**) that will interrupt close to the end of the sector transfer so that the DREQ **21***a* to DACK **21***b* timing may be determined on the last byte of the sector transfer.

Thus, it is possible to DMA shadow **58** all 512 bytes during a sector transfer, but that would cause the CPU to be totally consumed during the entire sector transfer time. The potential of losing processing activities elsewhere in the system are greatly increased, as in serial communications. Therefore, the clock interrupt routine **29***c* or method **29***c* of FIG. **5** may reduce the CPU involvement to a bare minimum during those floppy write operations with DMA Shadowing. Significantly, the timing may be adjusted to any number of bytes of a sector transfer, from a few bytes to the entire sector count.

One operation performed in the timer ISR routine **29***c* is to vector through the software decoding network **72** to the appropriate processing routine **84**, **86**, **88**. This process is illustrated graphically in FIG. **5**. The software decoding network **72** (software vector-table **72**) has one code point/entry **74**, **80**, **82** for each possible transfer byte in the sector.

The timer interrupt rate can now be in terms of 10's or 100's of byte transfer times. The vector table **72** may cause the program execution of the CPU **12** to enter a cascade **86** of DREQ **21***a*/DACK **21***b* checks only when the transfer (sector) will complete prior to the next timer interrupt. In short, the first entries **74** in the vector table **72** will return **84**, since another timer interrupt will occur before the sector transfer completes. The latter entries **80**, within the desired range, will cascade **86** from one DREQ **21***a*/DACK **21***b* detection to another (shadowing **58** the DMA transfers) until the last byte is transferred.

On the last byte being transferred, the data byte may be delayed by either activating a higher priority DMA **18** channel or masking the DMA channel of the FDC **20**. Although these two techniques are the simplest to program, numerous alternatives may be used to delay **60** data transfers on the DMA **18** channel of the FDC **20**, in accordance with the invention.

This software decoding network process **54** is the fastest known software technique for decoding and executing time-dependent situations. Space in the memory space **14** (e.g. the software decoding network vector table **72**) is traded for processing time, the amount of time it would take for one routine to subsume all functionality encoded in each of the

routines **84**, **86**, **88** vectored to through the software decoding network vector table **72**.

As indicated above, the entire software decoding network table **72** may be initially set to the address of an "exiting routine **84**." Then depending upon how slow or fast the system clock **22** interrupts, a certain number of the lower-indexed entries **80** of the table **72** may be set to the address of a processing routine **86**. These processing routines **86** may be identical and sequentially located in the routine **54**. Thus, the software decoding network vector table **72** may simply vector the timer ISR routine **29***c* within the driver **29***a* to the first of n sequentially executed processing routines. Here, n represents the number of bytes remaining in the sector transfer. In this way the last few bytes of the sector transfer can be accurately monitored (DMA Shadowing **58**) without significantly affecting overall system performance.

Each of the processing routines **86**, except the last one **88**, may perform exactly the same function. It is not necessary to be concerned with the timing between the DREQ **21***a* and DACK **21***b* signals until the very last data byte of a transfer. Therefore, the routines **86**, **88** above "shadow" **58** the operation of the DMA until the last byte (e.g. corresponding to entry **82** of the vector **72**) at which time the DMA channel of the FDC **20** is delayed as previously described.

Thus, through software DMA shadowing, it is possible to reliably determine when the last byte of the transfer is about to be transferred. Therefore, it is possible to force the last data byte's transfer to be delayed. An alternative approach may include a specialized application program **24** to control all aspects of the operation of the media drive **16**, e.g. floppy diskette drive **16**. This may include a transfer delay of a last byte, as indicated in FIGS. **6** (main application) and **7** (timer interrupt service routine). All aspects of the previous approach may be present. However, here they may be collected into a single application program **24** performing the required functions. The application program **24** may reprogram the system clock to interrupt at an accelerated rate and services the interrupt itself. The application program may then begin a repeated set of diskette write operations using the BIOS interface interrupt (O×13) and then read the written sectors back. Once the sector has been written and read back the data is compared to determine whether or not an undetected error has occurred. A running total of both detected and undetected errors may be output to a display.

Referring now to FIG. **6**, an application **24** may include steps **110**–**117**. Alternatively, preprocessing may begin at an entry point **100** leading to an initial command **102**. Command **102** is effective to request of a floppy diskette controller (FDC) **20** an identification. A status return of 0×90 (hexadecimal 90) should indicate that a FDC **20** is not of the type that is per se defective. However, faulty programming has now falsified the response fed to the test **104** in certain chips. Therefore, the command **102** may give rise to a false status return of 0×90 hexadecimal 90. This return does not guarantee that an FDC **20** is not defective.

Thus, a test **104** does not actually determine whether or not the status of an FDC **20** is defective. A negative response instead may be saved **105** to establish false negative responses programmed in by manufacturers. The display step **106** may include selected post processing to output results of the application **24**. Results may include, for example, an indication of whether the FDC **20** being tested is defective or not and whether a false negative response was given to test **104**, in view of the results of the test **112**. Accordingly, a status not equal to a hex **80** would ordinarily result in the test **104** signifying that an FDC **20** is not defective. The steps **105**, **106** thereafter verify defectiveness and improper circumvention of the test **104**.

After to the test **104**, the application **24** advances to a hook **110**. The hook **110** is effective to interpose a timer prefix **124** (see FIG. **7**) corresponding the prefix **34** of FIG. **3**, to be installed to operate at the beginning of a timer ISR **29***c* within the timer device driver **29***a*.

A test pattern **112** may format the last few (for example, 10) bytes of a sector write buffer **27***b*. Any known pattern may suffice, for example, a sequential list of all digits from zero to nine may be used. Importantly, the last two digits in such a sequence should be distinct. Thus, a string "0123456789" may provide a test pattern to be written in the last ten bytes of a sector. The test pattern may then be written from a buffer **27***b* to a medium **17** using the BIOS interface for the medium **17** and medium drive **16**.

Following the test pattern **112**, a test **114** may determine whether or not a write error has occurred in writing the buffer **27***b* to the medium **17**. A positive response to the test **114** results in an increment step **116**. The increment **116** tracks the number of successful detections of errors. Thus, the increment **116** indicates that another write error was successfully detected by the FDC **20**. Accordingly, the application **24** may advance from the increment **116** to a test **117**. A test **117** may determine the number of sectors to which the FDC **20** has attempted to write. If the response to the comparison of the test **117** is positive, then all tests are completed and the display step **106** follows. On the contrary, a negative response to this test **117** returns the application **24** to the test pattern **112**, initiating another test cycle.

A negative response to the test **114** indicates that a write error, known to exist, was undetected by the FDC **20**. Accordingly, a negative response to the test **114** advances the application **24** to a read **118**. The read **118** reads back the last previously written sector, using the BIOS diskette interface, such as the driver **29***b*. The step **118** may then increment the number corresponding to sectors that the FDC **20** has attempted to write.

The application **24** may next advance to the test **120** to determine whether the last byte that the read step **118** has read back from the written sector to a buffer **27***b* is the last, or the next-to-last element of the test pattern from the test pattern step **112**. That is, for example, in the example above, the test **120** determines whether or not the last byte read back to the buffer **27***b* from this sector being tested is correct (e.g. 9, a value other than 9 indicates that the FDC has failed to write the tenth element of the test pattern into the last byte location of the sector). This indicates that the FDC has not indicated a write error in the test **114**, and yet has produced the error detected by the test **120**. Thus, the last sector written is corrupted.

A negative response to the test **120** indicates that the last byte was not incorrectly written. Accordingly, the application **24** may advance to the test **117** to determine whether or not the testing is completed. A positive response to the test **120** results in an increment step **122**. The increment step **122** advances the count of undetected errors found during the operation of the FDC **20** during the testing in question. Thus, a step **122** results in a corruption count for sectors attempted to be written by the FDC **20**.

Referring now to FIG. **7**, and also cross-referencing to FIG. **6**, the hook step **110** may install a prefix **54** to a timer ISR **29***c* within the timer device driver **29***a* (see FIG. **4**). The hook **110** interposes the prefix **124** corresponding to the prefix **54** of FIG. **4**, after a call **125** or entry point **125** to the timer ISR **29***c* within the timer device driver **29***a*. Accordingly, whenever the timer ISR **29***c* within the timer device driver **29***a* is called, the prefix **124** will be run before any executables in the timer ISR **29***c* within the timer device driver **29***a*.

The prefix **124** may begin with a read **126** effective to determine a count corresponding to the number of bytes, or a countdown of the remaining bytes, being transferred by the DMA controller **18** from the main memory **14**, through the buffer **27***b* to the FDC **20**. The read **126** may also include a reading of a count (a tick count) of a timer **22** or system clock **22**.

Following the read **126**, a test **128** may determine whether or not an operation is in process affecting the FDC **20**. The FDC **20** is in operation if a count kept by the DMA controller **18** has decremented (changed) within an elapsed time corresponding to the maximum time required for a byte to be transferred. If no change has occurred during that elapsed time, then one may deduce that no activity is occurring. Accordingly, a negative response to the test **128** results in reexecution of the test **128**. Reexecution of the test **128** may continue until a positive response is obtained. Inasmuch as the application **24** is executing a write during the test pattern **112**, an eventual positive response to the test **128** is assured. In one embodiment of an apparatus and method in accordance with the invention, the first byte transferred may typically be detected.

A positive response to the test **128** advances the prefix routine **124** to a test **130** to test the countdown or count of the DMA controller **18**. The test **128** corresponds to detection of activity, whereas the test **130** corresponds to iteration of a shadowing process.

The test **130**, whenever a negative response is received, may advance the prefix routine **124** to the exit **138**.

On the other hand, a positive response to the test **130** advances the prefix routine **124** to a test **132** effective to evaluate whether or not the countdown is within some selected range at the end of a sector. A negative response to the test **132** indicates that the countdown is not within some desired end-of-sector range, so the prefix routine **124** should exit **138** without waiting longer. That is, interrupts will continue to occur with a frequency that will detect the desired range at the end of the sector being tested.

A positive response to the test **132** advances the prefix routine **124** to a test **134** for detecting the last byte to be transferred in a sector. If the DMA controller **18** is not counting the last byte to be transferred, then the test **134** may simply continue to test. When the countdown of the DMA controller **18** reaches a value of zero, a positive response to the test **134** advances to a delay step **136**.

The delay step **136** corresponds to the delay **60** illustrated in FIGS. **4**–**5**. The delay **136** may be implemented by preempting a channel over which the DMA controller **18** is communicating with the FDC **20**. For example, a first channel may be made active by some process, thus, overwriting communication over some channel having lesser priority, and corresponding to the FDC **20**. Likewise, the channel corresponding to the DMA communication with the FDC **20** may be masked (suspended) until the time elapsed for the transfer of the data to the sector has exceeded the maximum time permitted for such transfer. Thus, any and all opportunities for writing the last byte to the sector had expired. Thus, an error condition has been assured. Once the delay **136** has assured an error condition the exit **138** returns control of the processor **12** to the non-interrupted processing state.

The invention described heretofore provides detection solution that may be completely implemented in software as a device driver **29***b* that is capable of detecting defective FDCs **20** without visual inspection and identification of the FDCs. Furthermore, the unique and innovative approach taken, relying on DMA shadowing and use of a software decoding network, allows the implementation of the invention to accurately and correctly detect defective FDCs even when non-defective old-model FDCs are involved. Simply stated, it is not sufficient to determine whether the FDC under test is an old or new model FDC. Various vendors manufactured old-model FDCs that are not defective. Therefore, a two-phase detection process may correctly determine whether or not the FDC under test is defective.

The number of FDCs installed in computer systems today is well over 100 million. In order to identify defective FDCs vendors and consumers which have defective FDCs **20** installed have very few alternatives (e.g. recalls; replacement), of which most are extremely costly, for determining whether or not their systems are susceptible to the data corruption presented by defective FDCs **20**. Therefore, an apparatus and method that may be implemented as a software-only solution to this problem is a significant advance in the computer industry. Moreover, the robust design allows the apparatus and method of the present invention to dynamically adjust to processor speeds that encompass the original IBM Personal Computers executing at 4.77 MHZ to the latest workstations that execute at well over 200 MHZ.

The function of transfer of data to devices controlled by the FDC is described above in one embodiment as occurring through direct memory access (DMA). Nevertheless, this function may be accomplished with any suitable memory controller or other manner of data transfer. For example, one manner disclosed above involving a memory controller other than a DMA includes programmed input and output (I/O) accomplished directly by the microprocessor. Thus, in this instance, one skilled in the art will understand that in the discussion above, where a DMA is used, programmed I/O or other types of memory controllers may be readily substituted.

A further aspect of the present invention involves specific implementations of a method for detecting and preventing floppy diskette controller data transfer errors in computer systems. One embodiment of the method has been included by incorporating by reference U.S. Pat. No. 5,379,414 issued to Phillip M. Adams on Jan. 3, 1995.

One such specific implementation involves substituting a magnetic tape back-up device or an optical device, or other such peripheral, non-volatile memory device for the floppy drive previously described in the specific embodiment of the method. In so doing, the method substantially as disclosed is employed, substituting the use of the alternate peripheral, non-volatile memory device for the floppy drive. Appropriate commands are also substituted, as would be readily apparent to one skilled in the pertinent art.

The present invention may be embodied in other specific forms without departing from its spirit or essential characteristics. The described embodiments are to be considered in all respects only as illustrative, and not restrictive. The scope of the invention is, therefore, indicated by the appended claims, rather than by the foregoing description. All changes which come within the meaning and range of equivalency of the claims are to be embraced within their scope.

What is claimed and desired to be secured by United States Letters Patent is:

1. An apparatus for detecting a defective floppy diskette controller, the apparatus comprising:

a processor executing detection executables effective to determine an under run error undetected by a floppy diskette controller;

a memory device operably connected to the processor to store the detection executables and corresponding detection data;

a system clock operably connected to the processor to provide a time base;

a media drive comprising storage media for storing data formatted in sectors;

the floppy diskette controller operably connected to the media drive to control formatting and storage of data on the storage media; and

a memory controller operably connected with the floppy diskette controller for providing data to the floppy diskette controller.

2. The apparatus of claim **1** wherein the detection executables are effective to identify the floppy diskette controller.

3. The apparatus of claim **1** wherein the detection executables are effective to force an under run error.

4. The apparatus of claim **3** wherein the detection executables force the under run error by delaying a transfer of data to the floppy diskette controller.

5. The apparatus of claim **4** wherein the under run error comprises a delay in transferring a last byte corresponding to a sector involved in the transfer.

6. The apparatus of claim **4** wherein the detection data comprises a test pattern.

7. The apparatus of claim **6** wherein the under run error comprises the test pattern incorrectly copied onto the storage media.

8. The apparatus of claim **1** wherein the detection executables further comprise a prefix routine effective to hook a floppy device driver operating on the processor to control the floppy diskette controller.

9. The apparatus of claim **1** wherein the detection executables are integrated into an application directly loaded and executed on the processor.

10. The apparatus of claim **9** wherein the application is effective to determine on demand whether the floppy diskette controller is susceptible to undetected under run errors.

11. The apparatus of claim **1** wherein the detection executables include a shadowing executable effective to determine when a byte corresponding to a last byte of a sector is to be transferred.

12. A memory device operably connected to a processor, a memory controller, a floppy diskette controller controlled by the memory controller, and a media drive in communication with the memory controller, comprising:

a test pattern;

detection executables effective to be run on the processor to force and detect an under run error not detected by the floppy diskette controller, and

a readback buffer to store a copy of the test pattern read back from the media drive.

13. A method for testing controllers for controlling I/O to non-volatile memory devices, the method comprising:

providing a detection executable configured to interrupt a writing step of a controller;

delaying a transfer of a byte, corresponding to the writing step, for a time selected to cause an under run error in the transfer; and

verifying whether the controller detects an error in completing the writing step.

14. The method of claim **13**, further comprising selecting a non-volatile memory device from the group consisting of tape drives, magnetic drives, and optical drives.

15. The method of claim **13**, wherein the detection executable is configured to read a data count corresponding to the writing step.

16. The method of claim **15**, further comprising reading a data count, corresponding to the writing step, from a direct memory access controller's data transfer count register.

17. The method of claim **15**, further comprising reading a data count, corresponding to the writing step, from a CPU employing programmed input output.

18. The method of claim **13**, further comprising storing the detection executable and a corresponding detection data test pattern.

19. The method of claim **13**, further comprising increasing an interrupt rate corresponding to interrupting the writing step.

20. The method of claim **13**, further comprising causing and detecting a transfer corresponding to a last byte of a sector.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 6,401,222 B1 Page 1 of 1
DATED : June 4, 2002
INVENTOR(S) : Phillip M. Adams

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Title page,
The title is incorrect. Please delete "**DETECTION**", and insert therefor
-- **TESTING** --.

Column 1,
Line 59, please delete "NECK", and insert therefore -- NEC --.

Column 3,
Line 28, after "manifest itself", please insert -- . --.

Column 8,
Line 7, please delete "21 a", and insert therefore -- 21a --.

Signed and Sealed this

Twenty-fourth Day of December, 2002

[illegible]

JAMES E. ROGAN
Director of the United States Patent and Trademark Office

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 6,401,222 B1 Page 1 of 1
DATED : June 4, 2002
INVENTOR(S) : Phillip M. Adams

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

In the terminal disclaimer, please delete the first sentence "Petitioner, Phillip M. Adams, an individual residing at 325 North Federal Heights Circle, Salt Lake City, Utah, hereby represents that he is the owner of one hundred (100%) interest in the instant application", and insert therefore -- Petitioner, Phillip M. Adams, hereby represents that he is Managing Member of Phillip M. Adams & Associates, L.L.C., a limited liability company of the state of Utah having a principal place of business at 1460 Seville Way, Bountiful, Utah 84010 and a mailing address of P.O. Box 1207, Bountiful, Utah 84011-1207, and that Phillip M. Adams & Associates, L.L.C. is the owner of one hundred percent (100%) interest in the instant application --.

Signed and Sealed this

Fourteenth Day of October, 2003

[illegible]

JAMES E. ROGAN
Director of the United States Patent and Trademark Office